UPPSALA
UNIVERSITET

UPTEC E 13002
Examensarbete 30 hp
Juni 2013

An automatic voltage regulating system with Bluetooth communicating devices for brushless excitation of a synchronous generator

Eyuel Tibebu

Påbyggnadsprogrammet till civilingenjörsexamen i elektroteknik
Master Programme in Electrical Engineering

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

An automatic voltage regulating system with Bluetooth communicating devices for brushless excitation of a synchronous generator

Eyuel Tibebu

This thesis has been performed in order to earn a master's degree in electrical engineering. The task was to implement an automatic voltage regulator, AVR, to control the terminal voltage of the synchronous generator Svante at the Division of Electricity at Uppsala University. The AVR uses Bluetooth technology to transfer a control signal produced by a programmable logical controller, PLC, to phase-mounted SSRs that decides what proportion of the AC from a six-phase brushless exciter that is to be used for the magnetization of the rotor.
Test runs of the AVR were preformed with a regulator optimized according to the Ziegler-Nichols method and a static exciter that uses brushes and slip rings to apply voltage to the rotor winding. The results obtained complies with the assigned requirements set for this thesis and the AVR.
The primary focus of this thesis mainly lies in the construction of the control system, which include the programming of both PLC and Bluetooth communicating devices.

Handledare: Mattias Wallin
Ämnesgranskare: Urban Lundin
Examinator: Nora Masszi
ISSN: 1654-7616, UPTEC E 13002
Sponsor: ABB

Acknowledgements

I would like to thank my reviewer Urban Lundin and my supervisor Mattias Wallin for giving me the opportunity to perform my thesis in a field that I find extremely interesting. Your help and support along the way has truly been more than I ever could have hoped for.

Also, I would like to thank my co-worker and friend Boris Ivanić for making the long hours pass with ease.

Sammanfattning

Examensarbetet som utförts omfattar 30hp och är ett avslutande moment inom civilingenjörsutbildningen i elektroteknik.

Arbetet gick ut på att praktiskt framställa en automatiserad spänningsregulator för reglering av terminalspänningen hos synkrongeneratorn Svante vid Uppsala universitetets avdelning för elektricitetlära. Spänningsregulatorn använder Bluetoothteknik för att överföra en PLC-producerad styrsignal till fasmonterade SSR:er som styr hur stor andel av spänningen från en sexfasig borstlös matare som skall användas för magnetiseringen av rotorn.
Testkörning av spänningsregulatorn har utförts med en regulator optimerad efter Ziegler-Nichols metoden och en statisk matare som använder sig av borstar och släpringar för att förse rotorlindningen med ström. De erhållna resultaten överensstämmer med den ursprungliga kravbilden som var bestämd för AVR:en.

Fokus hos arbetet ligger huvudsakligen i konstruerandet av kontrollsystemet, vilket innefattar programmering av både PLC och Bluetoothkommunicerande enheter. Arbetet berör även andra övergripande delar av magnetiseringen.

Thesis Scope

The automatic voltage regulator, AVR, consists of several different parts, including programmable devices, Bluetooth communication, electrical connections, rectifying circuitry and much more. Even though the system as a whole was successful, there's still a lot of work that can be done on each individual part to obtain better results.
This master's thesis is only of 30 credit, thus limiting the time for any in-depth study when building something of this magnitude. Also, the AVR is to be part of an excitation system that features a six-phase brushless exciter which haven't yet been installed. Therefore, the results gained during test runs on the system can't be considered fully reliable.

Abbreviations

ACL - Asynchronous Connection Less
AFH – Adaptive Frequency Hopping
AT – Attention
AVR – Automatic Voltage Regulator
BT – Bluetooth
CFC - Continuous Function Chart
FBD - Function Block Diagram
HMI – Human Machine Interface
IDE - Integrated Development Environment
IEC - International Electrotechnical Commission
IL - Instruction List
ISM - Industrial, Scientific and Medical
LD - Ladder Diagram
PLC – Programmable Logic Controller
POU - Program Organization Unit
PWM – Pulse Width Modulated
RPM – Revolutions Per Minute
SCO - Synchronous Connection Oriented
SFC - Sequential Function Chart
SG – Synchronous Generator
SIG – Special Interest Group
SSR – Solid State Relay
ST - Structured Text
UART - Universal Asynchronous Receiver/Transmitter
USB – Universal Serial Buss

Contents

1 Introduction 1
1.1 Background 1
1.2 Project description 1
1.3 System overview 2
1.4 The experimental setup 3
2 Theory 5
2.1 Synchronous generators 5
2.2 Excitation systems 7
2.2.1 Exciters 7
2.2.2 Automatic voltage regulation 9
2.3 The PID controller 10
2.4 The Ziegler-Nichols tuning rule 11
2.5 Bluetooth communication 13
3 Hardware 16
3.1 PLC - PM564-t-eth 16
3.2 Control panel - CP620-web 16
3.3 Arduino 17
3.4 BTshield v2.2 18
3.5 Tillquist voltage measuring transducer 18
3.6 Current transducer - Honeywell CSLA1GD 19
3.7 SSR - Crydom MCPC1250A 19
3.8 Other 20
4 Software 21
4.1 PS501 Control Builder Plus 21
4.2 CoDeSys 21
4.3 Arduino 1.0 22
5 Method 23
5.1 Bluetooth connection / AT Commands 23
5.2 Programming the Arduinos 24
5.3 Custom made shields 26
5.3.1 Low-pass filter 26
5.3.2 Transistor inverter 26
5.3.3 Differential amplifying circuit 27
5.4 Programming the PLC 28
5.4.1 POUs 29
5.4.2 Visualizations 34
5.5 The control cabinet 35
5.6 The excitation system 36
6 Results 37
6.1 Applying the Ziegler-Nichols tuning rule 37
6.2 Controller performances - step responses 38
6.3 Manual operation 40
6.4 SSRs 40
6.5 The Bluetooth connection 40
7 Conclusions 42
8 Future work 43
9 References 45
10 Appendices 47

1 Introduction

1.1 Background

The energy that exist in flowing water has been used as a power source for centuries. From the old water wheels, to the present hydroelectric power plants, harnessing the energy out of running water plays an ever growing part in providing for our global electricity consumption. In a day when topics involving environmental issues and climate change is on everyones mind, substantial effort and money is put in to the research and development of renewable energy, with hydroelectric alternatives and especially hydropower paving the way.
Most hydropower plants use synchronous generators, SGs, to provide power to the grid. The grid-connected SG must be able to maintain a constant terminal voltage and compensate for disturbances that may cause it to differ from its desired voltage level. The SGs voltage level is monitored and kept stable by regulating the field current, I_f, applied to the rotor winding, increasing, or decreasing it until a desired terminal voltage is met. An external automatic voltage regulator, AVR, handles the regulating procedure. Together with an exciter, they form what is known as the excitation system.

1.2 Project description

At Uppsala University, the Division of Electricity has an experimental generator test setup for hydropower research. There, an excitation system including a six-phase brushless exciter and a Bluetooth communicating AVR is being installed. The primary focus of this thesis lies in the practical implementation of an AVR system that will wirelessly control the excitation, thus controlling the generator output voltage.

The AVR will regulate the voltage applied to the rotor winding of the SG “Svante” so that its terminal voltage is kept at a reference value of 156V. This will be done by running a variable proportional to the measurement of the terminal voltage through a regulating algorithm programmed in a programmable logic controller, PLC. The resulting control signal will then be transferred using Bluetooth technology to a receiving module on the rotating shaft that will distribute it as a control signal to six different switching relays. The relays pass on a controlled proportion of the AC voltage produced by the brushless exciter to a bridge rectifier that applies the newly rectified DC voltage to the rotor winding, causing a current flow that will magnetize the rotor poles.

Various programming environments and languages are used to program the PLC, the two Arduinos and to set up the Bluetooth connection.

1.3 System overview

Figure 1 displays the different devices included in the excitation system and how they interconnect with one another. Note that the figure displays the working principle of what is considered to be the finished product, including the brushless exciter which haven't been tested in this thesis.
A complete component walkthrough is listed bellow, explaining the role and purpose of the different devices.

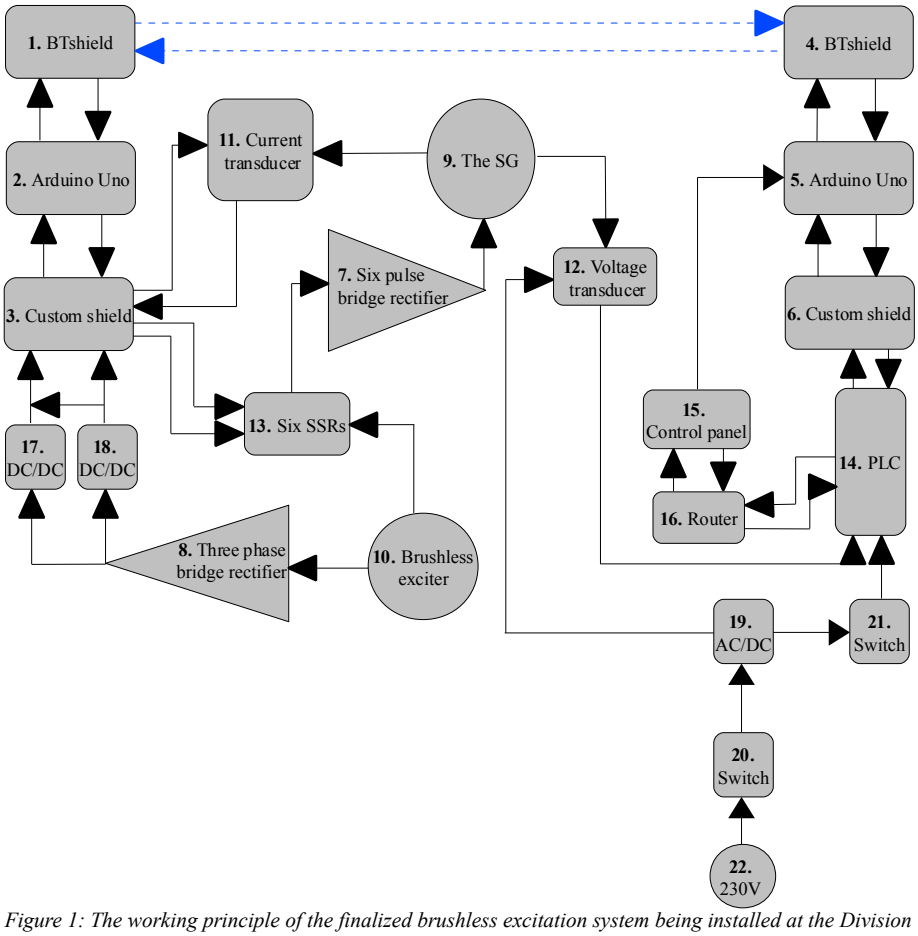


Figure 1: The working principle of the finalized brushless excitation system being installed at the Division of Electricity.

Component walkthrough:

1 A Bluetooth module called BTshield is configured as a slave and mounted on the rotating Arduino microcontroller board (2). The wireless connection established with the other BTshield (4) makes it possible to control the solid state relays, SSRs, (13) from a stationary position.

2 The rotating Arduino is programmed to handle the data received from the PLC and to provide feedback gathered from its inputs.

3 A custom made shield mounted on the rotating Arduino (2) that connects to the current transducer and the SSRs. The supply voltage needed to operate all rotating devices included in the excitation system is received

from two DC/DC converters (17,18).

4 The master BTshield is mounted on the stationary Arduino (5) inside the control cabinet. It exchanges data, including the regulated control signal, with the other BTshield (1).

5 The stationary Arduino microcontroller board is programmed much like the rotating Arduino (2). It handles the data being exchanged with the PLC (14) and Arduino (2). A universal serial buss, USB, cable connected to the control panel (15) provides power and must be disconnected whenever the stationary exciter is used.

6 A custom made shield for the stationary Arduino (5) that connects to the PLCs (14) inputs.

7 A six pulse bridge rectifier that converts the AC from the SSRs (13) to DC for the rotor field winding.

8 A three phase bridge rectifier that provides DC to the DC/DC (17,18) converters.

9 The SG.

10 The brushless exciter.

11 A current transducer that measures the field current I_f.

12 A voltage transducer measuring the terminal voltage of the SG (9) to provide the PLCs (14) regulating controller with the actual value $y(t)$.

13 Six SSRs, one on each phase of the exciter (10) receives the control signal from the PLC (14) to adjust the proportion of AC being fed to the six pulse bridge rectifier (7).

14 A PLC with web server functionality handles the regulating process.

15 A control panel to provide the user with human machine interface, HMI, functionality.

16 A router that enables wireless access to the HMI from a computer.

17 DC/DC converter that provides the custom made shield on the rotating side (3) with 12V+12V=24V.

18 DC/DC converter that provides the custom made shield on the rotating side (3) with 12V.

19 An AC/DC converter that provides the PLC (14) with 24V supply voltage.

20 On/off switch of the control cabinet.

21 An extra safety switch for the PLC (14), control panel (15) and the stationary Arduino (5).

22 230V power supply from a wall socket.

1.4 The experimental setup

The experimental setup available for conducting final tests of the AVR included a twelve-pole synchronous generator named Svante with a 75kW induction motor working as the prime mover. A 3:1 gearbox installed between the motor and the SG means that a rotational speed of 1500rpm is required from the motor in order for the SG to reach its synchronous speed.

Because the installment of the brushless exciter hadn't occurred at the time of this thesis, tests were performed with a static exciter that uses brushes and slip rings to supply a DC voltage to the rotor winding. The brushes and slip rings add to the resistance of the field winding, meaning that a higher voltage is required in order to produce the same level of field current as a brushless exciter [11].

Open-circuit characteristics

A voltage transformer rated 156:400 is connected to Svantes terminals, so that a voltage of 400V can be reached to enable synchronization with the grid. From the open-circuit characteristics shown in Figure 2, it can be seen that the field current provided by the excitation system should be at approximately 13A in order to produce a terminal voltage of 156V.

The open-circuit characteristics of the SG is obtained by increasing the field current during a no-load operation of the SG while rotating at its rated speed. Since no load is connected, measured voltages at the output will be equal to the internal voltage of the SG. The linear region of the plotted voltage-current relation displays where the generator is operating unsaturated [1].

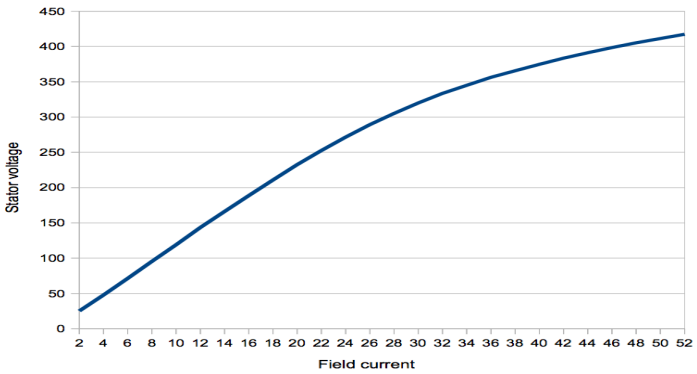


Figure 2: Open-circuit characteristics of the SG Svante.

Parameter	Value
Frequency	50Hz
Number of pole pairs	6
Speed	500rpm
Slots per pole and phase	3
Number of stator slots	108
Coil pitch	9
Stator inner diameter	725mm
Stator length	303mm
Air gap length	8.4mm
Power of driving motor	75kW
Rotor weight	900kg
Stator weight	700kg
Field winding (resistance without brushes or slip rings)	3.03Ω
Field winding (resistance with brushes and slip rings)	3.37Ω

Table 1: Parameter values of the SG Svante.

2 Theory

This chapter highlights the basic theory on which this thesis is based upon. It covers the principles behind a grid-connected SG and the different excitation alternatives that can be chosen to provide its field current. An automated regulating procedure of the field current is performed by an algorithm programmed into the AVRs PLC explained in the latter parts. Finally, an introduction to the Bluetooth technology is presented with a brief explanation of some of its most fundamental functionalities.

2.1 Synchronous generators

A hydropower plant uses SGs to supply the grid with electrical power. An SG consists of two fundamental parts, a rotor and a stator. The rotor magnetic field is created by a field current provided by the excitation system. It induces a voltage in the stator armature windings, converting the mechanical energy harnessed from the motion of water into electrical power.

Grid connecting

All SGs that are grid-connected are operating in parallel and rotating synchronously, this means that they all rotate with the same rotational speed. Because of the pure strength of a power-grid, some conditions has to be met before an SG is able to connect and contribute to it.

Fixed synchronization conditions:

- *A sinusoidal waveform*
  Method: A sinusoidal waveform on the output of the SG is the result of the construction itself.

- *Matched phase-sequence*
  Method: The phase-sequence is matched by pairing the terminal output of the SG to the grid in the right order.

In addition to the two conditions previously stated there are three conditions that must be met at the time of synchronization. These conditions are not “fixed” and therefore they have to be validated at each time of connection to avoid the risk of a large current flow in the junction due to voltage differences [1].

Non-fixed synchronization conditions:

- *Same voltages*
  Method: By increasing the field current, thus increasing the voltage induced in the armature windings. This is done until the terminal voltage, measured after a step-up transformer, equals the line voltage of the grid.

- *Slightly higher frequency on the connecting generator*
  Method: The electrical frequency, f_e, of an SG is determined by its number of poles, P, and the rotational speed of the rotor magnetic field, n_m, set by the prime mover, see equation (1).

$$f_e = \frac{n_m P}{120} \tag{1}$$

  The frequency of the generator is set slightly higher than the grids to ensure that the generator will supply power instead of consuming it (acting as a motor) during connection.

- *Same phase angles*
  Method: After the frequency is set, voltages in the interacting systems will change phase relative to each other in a slow manner. A frequency meter observes the changes until the phase angles are equal.

A grid-connection is possible once all conditions have been met.

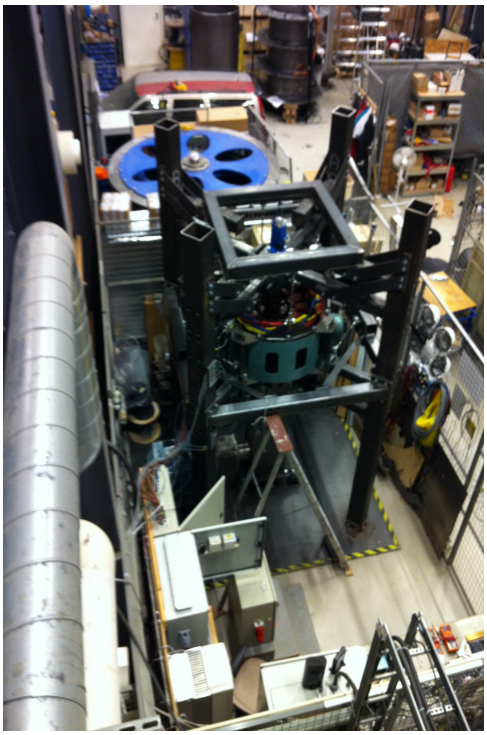


Figure 3: The twelve-pole SG "Svante" must rotate at 500rpm to match the grid-frequency 50Hz.

2.2 Excitation systems

Most generators need excitation systems to generate the magnetic field necessary to induce a voltage in the stator. Excitation systems consists of an exciter and an AVR working to supply the SGs rotor winding with a field current, I_f, that will magnetize its poles. A voltage V_f coming from the exciter is applied to the rotor winding, creating the current flow I_f, while the AVRs role is to control what voltage level the exciter should put out by reading various inputs from the excitation system and passing on a control signal. One can think of the AVR as the controlling device of an electrical valve between the exciter and field winding, adjusting the amount of current that is used to magnetize the rotor [2,3,4].

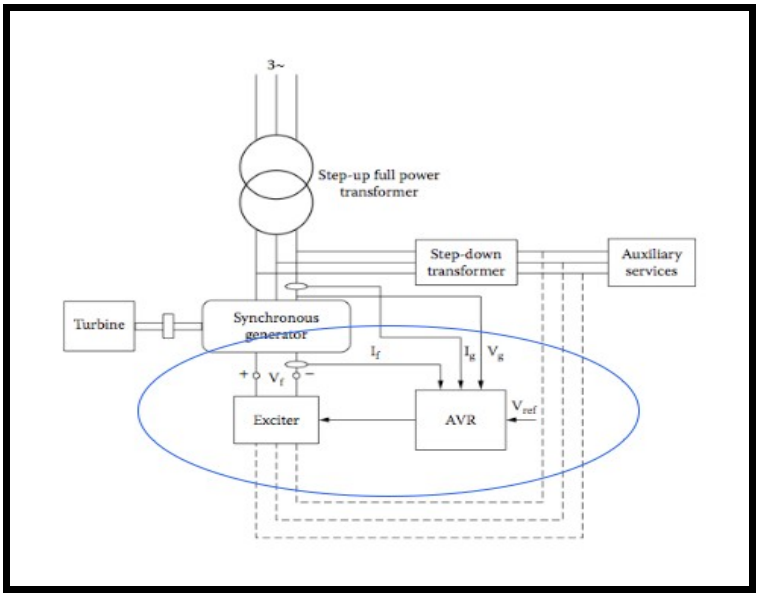


Figure 4: The blue ellipse encloses the excitation system of a power plant, including a static exciter and an AVR [2].

2.2.1 Exciters

There are three different types of exciters:

- DC exciters
- AC exciters
- Static exciters

DC exciters

The DC type of exciters were a favorable choice during the years leading up to the mid-1960s, before they successively got replaced by the AC exciters, however they still exist for many SGs rated lower than 100 MVA. They can be either self-excited or separately-excited and kept rotating by the same shaft as the SG or by an external motor.
The exciter uses DC generators to produce the DC voltage for the SGs rotor field winding. One might think that a DC generator produces a DC voltage, but in reality they produce AC.

DC exciters use commutators to reverses the flow of current periodically, which means that the current that runs through an externally connected device, only moves in one direction. The excitation system supplies the SG with the DC voltage with the help of brushes and slip rings.

AC exciters

AC exciters use SGs that more or less can be described as turned inside out, to feed the main generator with its field current. A magnetic field is generated by the stator while the output is collected off the rotor winding. The generator produces an AC voltage while rotating on the same shaft as the main SG. The voltage then gets rectified, turning it into DC so that it can be used to magnetize the rotor poles.
The rectifying system can be setup in different variations, all carrying their own pros and cons, see examples bellow.

The stationary non-controlled rectifying system can't directly control the field current of the SG, thus the AVR output must directly control the strength of the exciters magnetic field to enable an adjustable terminal output on the SG. Since it is a stationary system it connects to the SG with brushes and slip rings.

A stationary controllable rectifying system uses thyristors instead of stationary diodes and can therefore control the exciter output directly. The AVR triggers the thyristor gates to make them conduct, providing the system with a better response time in comparison to systems that are non-controllable. Slip rings and brushes are used to apply the SG field current.

The rotating rectifying system, often referred to as a *brushless excitation system* is mounted on the same shaft as the SG, eliminating the need of slip rings and brushes by connecting its output directly to the SGs field winding. Because the system is rotating, the AVR controls the SGs terminal voltage by regulating the exciters magnetic field see Figure 5.

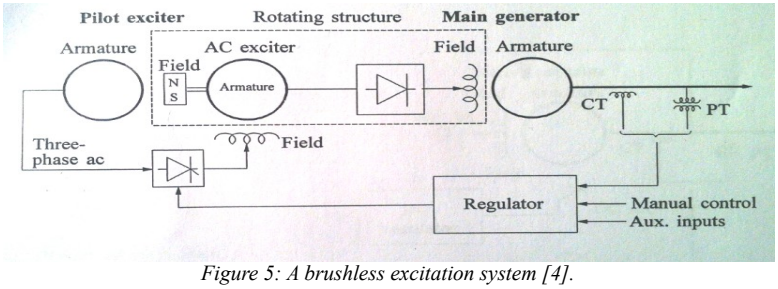


Figure 5: A brushless excitation system [4].

Static exciters

A static exciter is a completely stationary exciter that doesn't include any moving parts. The most commonly form of this type of exciter consists of a static three-phase bridge rectifier that is fed power from the terminal output of the SG through a step down transformer, meaning that unlike other types of exciters they don't feature an external generator. The rectifier, either controllable or non-controllable, supply field current to the SG with the help of brushes and slip rings.

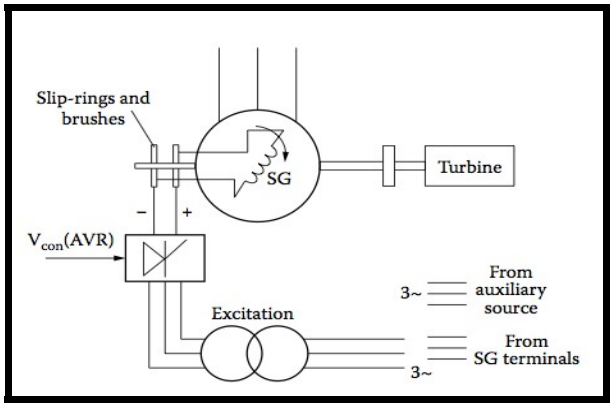


Figure 6: A static excitation system [2].

2.2.2 Automatic voltage regulation

The job of the AVR is to ensure a constant and stable terminal voltage on its connected generator. If the load condition changes, the AVR must regulate the excitation output to compensate for the new demand. This is done by running the error between the terminal voltage and a reference value through a regulating algorithm that is programmed in a logical controller.

PLC – A brief introduction

In the years before 1968, electromechanical processes especially used by car manufacturers were controlled by relay based equipment. This equipment would consist of hundreds and sometimes thousands of parts that had to be updated yearly due to car-model upgrades. An update of a control system was both costly and time-consuming and had to be done manually on site.
The introduction of the minicomputer in the 1960s gave birth to the idea of automating electromechanical process with the use of programable computer-based devices, the Programable Logic Controller or the PLC as it is abbreviated had been born.
In the beginning the PLCs were programmed in Ladder Diagram to resemble the schematic outlining of relay logic so that it wouldn't demand a rich programming background in order to use it. Since then its programming environment has evolved to include a variety of languages and the usability nowadays covers motion control, sequential relay control, process control, networking and much more, making it a very useful tool in the modern day industry [26].

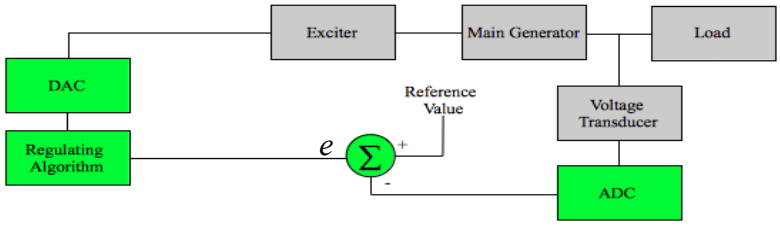


Figure 7: The green boxes displays the role of the PLC within an excitation system.

2.3 The PID controller

A feedback controller makes adjustments to the input of a system with the purpose of correcting any eventual errors that it might have on its output. If a system output deviates from its target value the controller output either increases or decreases, attempting to steer it back [2, 5].

The control error

The control error, *e(t)*, is the difference between the reference value, *r*, which is the desired value of the system output, and the actual value, *y(t)*, that the output produces.

$$e(t) = r - y(t) \tag{2}$$

Proportional action

This action sends out a linear control signal, *u(t),* that is proportional to the control error by a factor determined by the proportional gain, k_p, whilst in the proportional band. The proportional band represents all control errors for which the controller is linear, outside it, the output gets unstable.

$$u_P(t) = k_p e(t) \quad if \quad e_{min} \leq e \leq e_{max} \tag{3}$$

Integral action

The proportional action is in many aspects limited due to deviations between the actual value and the reference value during steady-state, also known as the steady-state error.
With an integral algorithm the output of the system is guaranteed to be equal to the reference during steady-state, it does however not guarantee a steady-state in itself. The integral action sums up all of the instantaneous errors over time and multiplies them with its integral gain, k_i, to create its present output.

$$u_I(t) = k_i \int_0^t e(\tau) d\tau \tag{4}$$

When combined with the proportional action, they form the PI-controller seen in equation (5), that can be parameterized differently using the integral action time, T_n, according to the relation presented by equation (6).

$$u_{PI}(t) = k_p e(t) + k_i \int_0^t e(\tau) d\tau \tag{5}$$

$$k_i = \frac{k_p}{T_n} \tag{6}$$

Derivative action

While the integral action compensates for the errors in the past, the derivate action aims to compensate for the errors yet to occur [5]. This is done by mathematical extrapolation, using the tangents of the error curve multiplied with the derivative gain, k_d, as seen in equation (7).

$$u_D(t) = k_d \frac{de(t)}{dt} \tag{7}$$

The three actions, proportional, integral and derivative are the building blocks of the PID feedback controller, see equation (8).

$$u_{PID}(t) = k_p e(t) + k_i \int_0^t e(\tau)\, d\tau + k_d \frac{de(t)}{dt} \tag{8}$$

$$k_d = k_p T_v \tag{9}$$

A different parametrization of the PID controller can be made using the integral action time, T_n and the derivate action time, T_v, according to the relations presented by equations (6) and (9).

$$u_{PID}(t) = k_p \left(e(t) + \frac{1}{T_n} \int_0^t e(\tau)\, d\tau + T_v \frac{de(t)}{dt} \right) \tag{10}$$

Equations (8) and (10) are both the same PID controller, though they are parametrized in different ways. All other controllers can be obtained from these two equations by changing the parameter values, thus excluding any non-desirable actions, see Table 2.

	Equation (8)			Equation (10)		
	k_p	k_i	k_d	k_p	T_n	T_v
P	value	$\rightarrow 0$	0	value	$\rightarrow \infty$	0
PI	value	value	0	value	value	0
PD	value	$\rightarrow 0$	value	value	$\rightarrow \infty$	value
PID	value	value	value	value	value	value

Table 2: Parameter values of different controllers.

2.4 The Ziegler-Nichols tuning rule

There are different methods to produce the necessary parameters k_p, k_i and k_d for the different controllers. By using computer software, mathematical models or

manual tuning the parameters can be solved with different degrees of efficiency. The mathematical approach produces the most preferable results, it is however a tough task to tackle due to the fact that the model of a control system often is far to complex and time-consuming to solve. Not all methods are suitable for every system, therefore by ascertaining whether the system has to be active in order to find the necessary parameters, is a major key in deciding which method to apply [6, 25].

According to Ziegler-Nichols, the system is stable if the subsequent amplitudes of the waves generated by a step change of disturbance, v, have a ratio not greater than ¼ after using their method, see Figure 8.

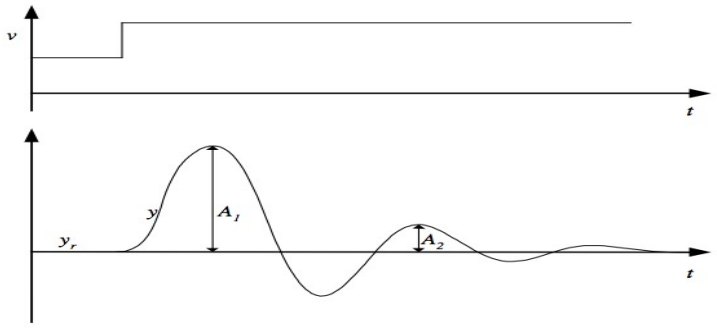


Figure 8: Subsequent waves must not be greater than ¼ according to Ziegler-Nichols [25].

The method is performed by finding the ultimate sensitivity, s_u, and the ultimate period of oscillation, p_u.
These two variables are acquired when increasing the k_p parameter of a P-controller to the point of sustained oscillations at the output, at which point $s_u = k_p$ and $p_u = p_{u2} - p_{u1}$, note that the oscillations must stay within the systems maximum and minimum range for the Ziegler-Nichols method to be applicable.

	k_p	k_i	k_d
P-Controller	$0.5s_u$	-	-
PI-Controller	$0.45s_u$	$1.2k_p / p_u$	-
PID-Controller	$0.6s_u$	$2k_p / p_u$	$k_p p_u / 8$

Table 3: Ziegler-Nichols

From the basis of Ziegler-Nichols tuning-rule, other methods have been developed to estimate the necessary parameters of a PID-controller for certain purposes when the shape of the output is vital for the system, see Table 4.

	k_p	k_i	k_d
PID with no overshoot	$0.2s_u$	$2k_p / p_u$	$k_p p_u / 3$
PID with some overshoot	$0.33s_u$	$2k_p / p_u$	$k_p p_u / 3$
PID with Pessen integral rule	$0.7s_u$	$2.5k_p / p_u$	$0.15k_p p_u$

Table 4: PID parameters of different tuning methods.

2.5 Bluetooth communication

The technology behind Bluetooth was invented in 1994 by engineers at the Swedish company Ericsson. In 1998 they formed a collaboration with other companies, assigning a special interest group, SIG, to further develop the Bluetooth technology with the intent of creating a new way to interconnect their products without the need of wires. Nowadays Bluetooth communicating devices can be found in almost all portable electronic equipment, from your mobile-phone to your laptop.
Six key functionalities of Bluetooth communication are spectrum, frequency hopping, packages, networks, physical links and energy conserving modes, a short introduction is listed bellow [13, 14].

Spectrum

Bluetooth uses the industrial, scientific and medical, ISM, radio band, which is unlicensed in most countries, at 2.4 to 2.485 GHz when operating.

Frequency hopping

The ISM band is widely used by many appliances, therefore, Bluetooth utilize adaptive frequency hopping, AFH, at 1600 hops/sec to constantly search for available frequencies within the spectrum to work on, thereby minimizing the risk of running in to interference.

Packages

Each package contains three parts, a 72-bit access code, a 54-bit packet header and payload that can vary from 0- to 2745-bits. The access code is used to pinpoint other devices that share the same network and to grant them access to the package content. The header holds the address of active recipients along with information of the package duration and the error-correction scheme, while the actual data that is to be transferred is carried by the payload.

LSB 72	54	0-2745 MSB
Access code	Header	Payload

Figure 9: Structure of a Bluetooth data package.

Networks

When Bluetooth devices are connected they can communicate with each other through short-range, ad-hoc networks called Piconets. In every Piconet there is one device that holds the role of master. The master device sets the conditions for the network by determining the access code and the clock for the frequency hopping sequence. A Piconet network can include up to seven different devices excluding the master. These devices are known as slaves and they operate by the conditions set by the master. A device belonging to a Piconet can coexist in another Piconet network simultaneously, independently of what role it plays, meaning that a slave in one Piconet can figure as a master in another. When multiple Piconets interlace it is called a Scatternet, see Figure 10. Beacause of the

frequency hopping, a Scatternet can consist of up to ten completely full Piconets before the networks overload and saturation occurs.

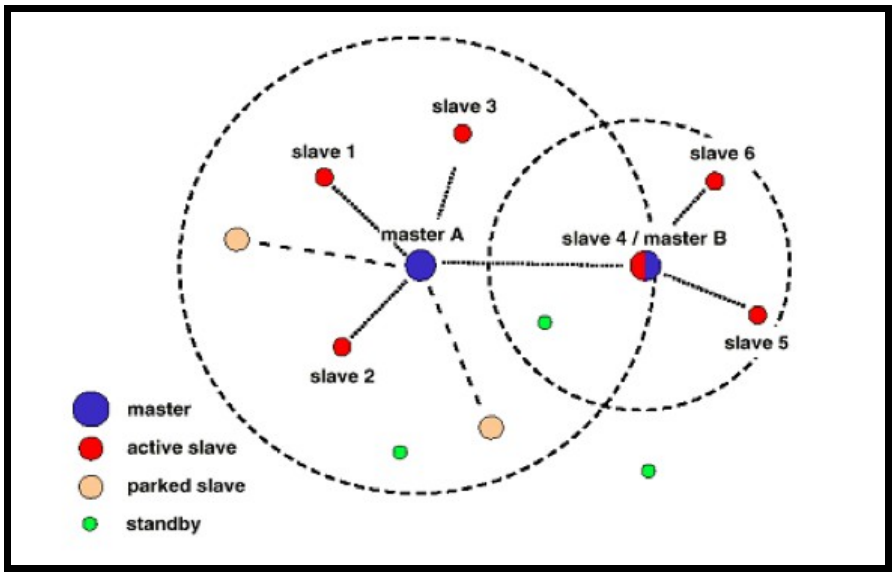


Figure 10: A Scatternet consisting of two interlacing Piconets [14].

Physical links

The synchronous connection-oriented, SCO, and the asynchronous connection-less, ACL, are two different connections that Bluetooth utilizes to form the links between a master and a slave. Each unit can have one ACL link and two SCO links, three if the unit is of master character, working at the same time.

SCO – Channel that handles voice transfers with a link working at 64kb/s in both directions.
ACL – Uses the slots not occupied by the SCO to send and receive data between master and all slaves. A master can broadcast to all slaves at once or talk to them separately through ACL.

Modes

One of the greatest strengths of Bluetooth technology is the low power consumption required during different operational modes.

- Active
  When a module is in this mode it's actively taking part of a physical link, consuming around 40mA to 50mA in the process. The master polls the slave to enable them to send packages and re-synchronize.

- Sniff
  A low consumption mode where the slave only listens to the Piconet during brief instants of time, reducing the power consumption to this time-window.

- Hold
  Another low consumption mode that consumes less power than the sniff mode. An ACL connection between two devices can be put in hold mode if there isn't any need for their transceivers to be active during a certain time-period. A counter in the Bluetooth module is the only active part while hold mode is running. The module resumes to active mode after the counter has reached its target time.

- Park
  A module in park mode has given up its MAC address and is no longer an active participant of a Piconet network. It is however still possible to both listen to broadcasts and to re-synchronize to the master.

3 Hardware

As seen in Figure 1, an excitation system consists of several interacting components, most of which belonging to the AVR. This chapter presents an introduction of all the major devices and components used to build the AVR system.

3.1 PLC - PM564-t-eth

The PLC is found at the heart of the AVR, programmed with the necessary features like, controllers, memory storage, HMI visualizations and measurement indicators, see component 14 in Figure 1.

The PM546-t-eth is a PLC donated by ABB and belongs to the stand-alone branch of their AC500 series that is easily programmable using ABBs CoDeSys based system PS501 Control Builder Plus. It has a programming processing time of 0.8µs/instruction and like other PLCs it provides a reliable performance when dealing with automated processes. Aside from its processes handling it can be programmed using CoDeSys visualizations in order to enable HMI web server functionalities that will be accessible through web applications.

Figure 11: The PLC PM564-t-eth donated by ABB [15].

Main characteristics:

- 24VDC or 100-240VAC power supply
- A 128kB program memory
- Ethernet interface
- 6 digital inputs
- 2 analog inputs (configurable as digital inputs)
- 1 analog output

3.2 Control panel - CP620-web

A user can gain access to the web interface provided by the PLC from the MicroBrowser application installed on the control panel, see component 15 in Figure 1.

Figure 12: The control panel CP630-web utilizes its pre-installed web browser to access the user interface [16].

The CP620-web is a color 4.3" touchscreen display that is running a Microsoft Windows embedded operating system. The MicroBrowser installed is used to communicate with the PLC by accessing the web server content from its ip address [12].

3.3 Arduino

Arduino is an open-source electronic prototyping platform based on a programable microcontroller board with its own programming environment for writing software. It can be used to control various objects like lights and motors or to sense the input from switches or sensors.

Figure 13: The Arduino logo [7].

The company started out 2005 in Italy and their boards has since then become a popular choice for your everyday electronics hobbyist. They supply numerous boards with a broad range of capabilities for a low cost. With their exposed connectors, compatible shields can be used to enhance the capabilities even further [7].

Arduino Uno Rev3

Two microcontroller boards, one in the control cabinet and the other on the rotating shaft are programmed to send and receive information from and to the PLC, as well as between one another, see components 2 and 5 in Figure 1.

The Uno can be seen as the flagship of Arduino boards. It comes based on the ATmega328 microcontroller and with the Atmega16U2 preprogrammed as a USB-to-serial converter. The ATmega328 makes it possible to communicate with the Arduino by universal asynchronous receiver/transmitter, UART, serial communication from a computer or other UART compatible computer hardware. The UART breaks up bytes into bits before sending the data out in a serial stream on the transmitting pin (pin1). When serial data is being received (pin0) the exact opposite procedure occurs and the UART reassembles the incoming bits back into bytes.
Because the Uno connects to the computer by USB, the serial data flow is being converted by the ATmega16U2 before reaching the UART of the ATmega328 [7].

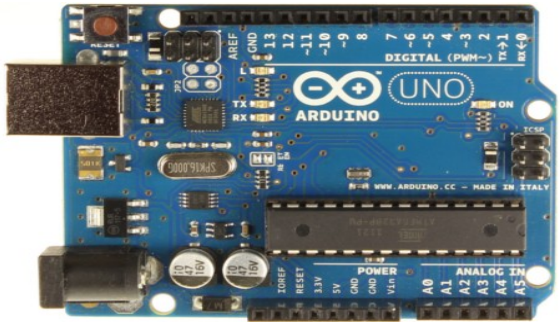


Figure 14: The Arduino Uno [7].

The Arduino Uno has 6 analog 10-bit inputs (A0 to A5) and 14 digital I/Os (pin0 to 13), 6 of which are capable of delivering an 8-bit pulse width modulated, PWM, signal. The I/Os are bound by a 5V upper limit and capable of delivering or receiving 40mA, however the voltage-range of the analog inputs can be adjusted by a reference voltage on the AREF pin. The Uno is powered by either the USB-cord (5V), the Vin pin (7-12V) or the DC power jack (7-12V).

3.4 BTshield v2.2

The two BTshields, one on each Arduino, create a Bluetooth connection where information between them both can flow, se components 1 and 4 in Figure 1.

The BTshield v2.2 is an Arduino compatible master/slave-shield created by IteadStudios for wireless data transfers using UART and Bluetooth 2.0. The shield basically consists of the Bluetooth serial module HC-05 that directly connects to the Arduino pins for power and communication. The HC-05 module is programmed directly by serial communication via its TX and RX ports using attention, AT, commands from a terminal window on a computer. The Bluetooth module can be configured in numerous ways depending on the conditions in which the module is to be used. The option to make the shield into a master or a slave device, to bind it to other specific devices or to make it listen for inquires from its surroundings are all settings that are configurable when programming the HC-05 [17].

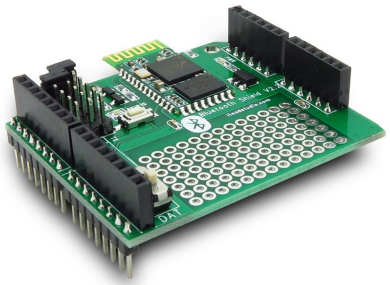


Figure 15: The BTshield v2.2 [18].

Figure 16: The HC-05 Bluetooth chip [19].

Aside from the all important HC-05, the shield consists of a mode switch, a status LED, a UART multiplexer, a reset button and some free area to add external components if needed.
The mode switch is connected to the PIO11 input of the HC-05 which sets the Bluetooth module in either command mode, where it listens and responds to AT commands, or in data mode, where it's non-responsive to AT commands.
The UART multiplexer makes it possible to choose different pins on the Arduino aside from pin0 and pin1 (RX and TX) to run serial communication with the BT shield. By having this alternative the Arduino UART is kept isolated, making it possible to utilize the USB for serial communication with a computer at the same time as the BTshield is communicating with the Arduino.

3.5 Tillquist voltage measuring transducer

The AC line voltage of the SG is measured by the voltage transducer which produces a proportional DC signal that is fed to the PLC, see component 12 in Figure 1.

The in- and output of the transducer are galvanic separated due to a transformer at the input. The scaled voltage on the secondary side of the transformer runs through the true-rms to dc converter and the output amplifier, resulting in a DC voltage proportional to the AC input [20].

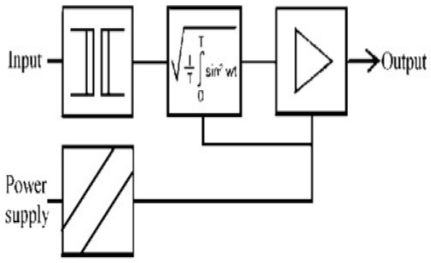


Figure 17: Block schematic of the working principle of the measuring transducer [20].

3.6 Current transducer - Honeywell CSLA1GD

Measures the field current, I_f, in the rotor winding and produces a proportional DC output to the rotating Arduino, see component 11 in Figure 1.

When current flows in a wire, a magnetic field proportional to it, is created around its center. The Honeywell current transducer CSLA1GD works according to this principle, measuring the strength of the magnetic field produced by the wire that runs through its hole. An analog DC representation between 0V and the half of its supply voltage (Vss/2) is then produced on the output.

Figure 18: A Current transducer [21].

3.7 SSR - Crydom MCPC1250A

Six different SSRs receive a control signal from the AVR to adjust the level of AC being fed to the rectifying circuit, see component 13 in Figure 1.

The SSR is an electronic switching relay that works without the need of moving parts.
The MCPC1250A is a proportional SSR with microprocessor driven phase-angle control. An analog DC input-signal between 0 and 5 volts controls the power supplied to its load from 0 to 100% of an AC voltage.

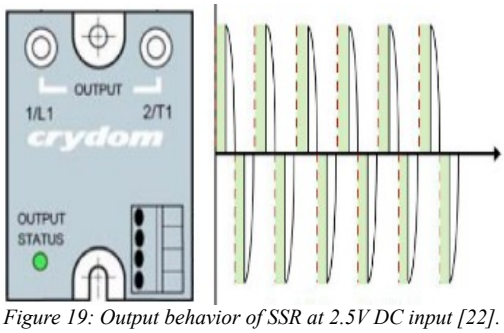


Figure 19: Output behavior of SSR at 2.5V DC input [22].

3.8 Other

Operational amplifier - OPA337

Part of the differential amplifying circuit on the rotating custom made shield that enhances the resolution of the current transducer, see component 3 in Figure 1.

Router – Netgear CG3100

Enables computers to wirelessly connect to the system through Wi-Fi, see component 16 in Figure 1.

Rectifier diode modules

Will form a six-phase rectifying bridge on the rotating shaft to provide the rotor winding with a DC voltage, see component 7 in Figure 1.

Traco AC/DC power supply

Supplies 24V DC to the PLC, see component 19 in Figure 1.

Traco DC/DC converter

Two converters in series supply 12V and 24V to the custom made shield on the rotating Arduino, see components 17 and 18 in Figure 1.

Three-phase bridge rectifier – DB15-04

Rectifies the AC voltage from three of the phases belonging to the six-phase brushless exciter in order to supply the the two DC converters with a DC voltage, see component 8 in Figure 1.

Figure 20: The three-phase bridge rectifier on the left and the two DC/DC converters on the right supply the rotational custom made shield with 12V and 24V.

4 Software

An extensive amount of time have been spent programming the PLC and the two Arduinos included in the AVR system. While Arduino uses their own software to program its different boards, the PLC uses two separate ones, one to setup its characteristics and the other for code generation.

4.1 PS501 Control Builder Plus

The ABB software PS501 Control Builder Plus combines the different tools necessary in order to configure your automation project. When starting a new project, the desired PLC is selected from an internal list, which configures the characteristics of the project to fit the selected model. After a PLC model is selected the user can edit several different settings like the I/O configuration and the ip address.
Once a project is created with Control Builder Plus the CoDeSys software can be launched from the AC500 icon found in the sidebar [10].

4.2 CoDeSys

CoDeSys is a development environment based on the IEC standard, it stands for Controlled Development System and is used for PLC code generation.
The first program organization unit, POU, automatically gets handed the name PLC_PRG which sets the process in motion, much like the main function in a C program. Now that the process has started, other POUs like programs, functions or function blocks can be written to the project, while the PLC_PRG acts as an outer shell holding POUs together. Every POU consists of two parts, a declaration area where variables, IOs, and imported elements are declared and a body, where the writing of the actual program takes place [8].

A body is written in one out of the six languages listed below. The first five are part of an international electrotechnical commission, IEC, supported classification while the sixth one is well suited for feedback applications.

Textual languages:

- IL – Instruction List
  IL is a primitive language, not used by many today. Low-level languages An IL-code consists of a series of instructions, each on a new line. The instruction lines are piled upon one another to resemble a list form hence the name IL.

- ST – Structured Text
  ST is a high-level language, much like C, C++ and Java, meaning it is built to look like a human language or a mathematical notation, making it easy to both use and understand.

Graphical languages:

- SFC – Sequential Function Chart
  The graphical oriented language SFC makes it easy to graphically visualize step-sequences of running processes. A POU written in SFC mainly consists of, steps, actions and transitions. A step have the form of a block and are connected to each other with transition-lines that become active once a transition condition of boolean type is fulfilled. Actions are placed inside the steps and contain instructions that can be written in IL, ST, FBD, LD or SFC.

- FBD – Function Block Diagram
  When using FBD, signals are passed through function blocks from left to right. The function blocks can either be self made with input and outputs or from the internal libraries, like logical expressions and mathematical operations. Notice that to connect two function blocks the input and output of the interacting blocks must be of the same type.

- LD – Ladder Diagram
  The purpose of LD is to give a graphically representation similar to an electric circuit. It is suitable when creating logical switches and calling other POUs.

- CFC – The Continuous Function Chart Editor
  CFC is much like the FBD but with freely placable elements, which means it can handle feedback.

4.3 Arduino 1.0

The Arduino software has evolved from the software of another open-source electronic prototyping platform called Wiring. Wiring itself is based on the programming language Processing and the Processing integrated development environment, IDE. Arduinos programming language is a simplification of the C language, created to make the programming process easy for a beginner. When a program, or a sketch as it is called for Arduino gets uploaded, the written code is translated into C, meaning that any devices running an Arduino sketch actually is programmed in C.
The Arduino IDE provides an easy to use interface that is pretty straight forward when it comes to writing and uploading. There are however two settings that has to be correct in order to preform an upload. Once the correct Arduino board and the correct serial port have been chosen from the tools menu, the sketch will be ready, provided that the code is compiled without errors [7, 9].

5 Method

This chapter describes the construction of the AVR system. It aims to provide the reader with the information necessary to understand the underlying structure of the sketches and programs that have been written for the two Arduinos and the PLC so that future changes may be easily implemented. Aside from the programs, this chapter includes the construction of two custom made shields and an overview of the excitation system used to preform the test runs.

5.1 Bluetooth connection / AT Commands

The two BTshields need to be configured so that communication between them is possible. By binding the addresses of the two shields they will connect to each other if they are within a 10m radius.

To program a shield is a step by step process:

1. The shield is placed on an Arduino. It is important that the Arduino itself isn't programed with any serial running code!
2. The jumpers on the UART multiplexer are placed so that the transmitting and receiving pins of the shield connect to the atmega16U2 USB-to-serial converter on the Arduino, see Figure 21.

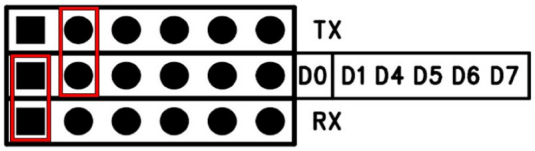


Figure 21: Jumper position of the UART multiplexer during configuration.

3. Connect the Arduino and the stackable shield to a computer and run a terminal program to gain access to the HC-05. The BTshield is preset to work at a baud rate of 38400 which means that the terminal program must be configured to the same speed in order for communication with AT commands to be possible.
4. Shift the mode switch to CMD and type in “AT” in the terminal window. If the HC-05 responds with an OK message it means that it is responsive to programable AT commands.
5. Acquire the device address with the command “AT+ADDR?” .

Once these steps are completed for both shields they are configured as master and slave and set to search for each other using their counterparts Bluetooth address. The configuration is made with other AT commands shown in Figure 22 and 23.

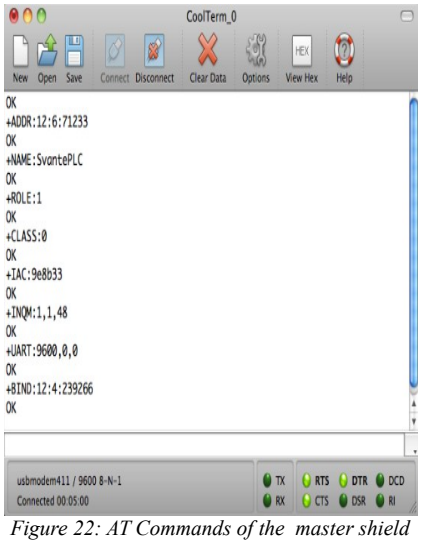


Figure 22: AT Commands of the master shield that connects to the stationary Arduino.

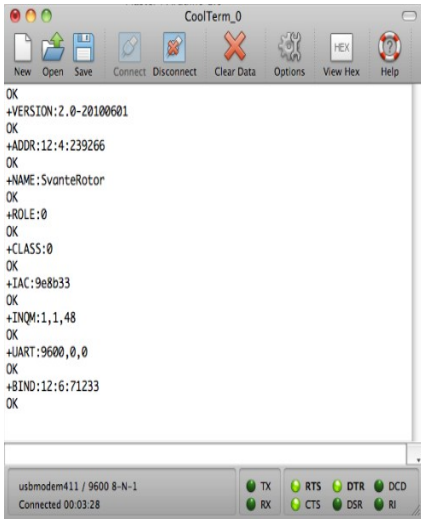


Figure 23: AT Commands of the slave shield that connects to the rotational Arduino.

As seen in Figure 22 and Figure 23 the baud rate of both shields have been decreased to further reduce the risk of data being lost during wireless transfers. The "AT+ROLE" command decides which role a device should assume, master or slave. SvantePLC is located in the control cabinet of the AVR, it holds the role of the master and uses "AT+BIND" together with the address of the slave to setup the connection.

5.2 Programming the Arduinos

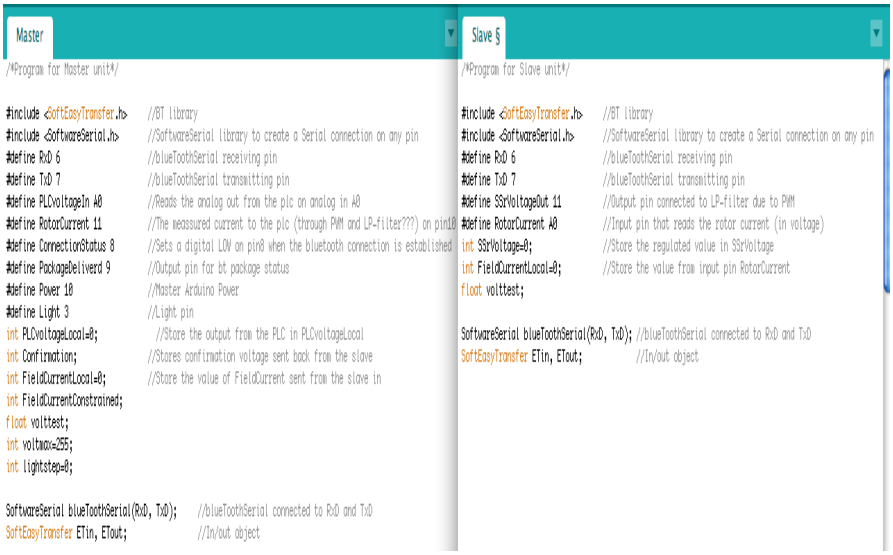

Master
/*Program for Master unit*/
#include <SoftEasyTransfer.h> //BT library
#include <SoftwareSerial.h> //SoftwareSerial library to create a Serial connection on any pin
#define RxD 6 //blueToothSerial receiving pin
#define TxD 7 //blueToothSerial transmitting pin
#define PLCvoltageIn A0 //Reads the analog out from the plc on analog in A0
#define RotorCurrent 11 //The meassured current to the plc (through PWM and LP-filter???) on pin10
#define ConnectionStatus 8 //Sets a digital LOW on pin8 when the bluetooth connection is established
#define PackageDeliverd 9 //Output pin for bt package status
#define Power 18 //Master Arduino Power
#define Light 3 //Light pin
int PLCvoltageLocal=0; //Store the output from the PLC in PLCvoltageLocal
int Confirmation; //Stores confirmation voltage sent back from the slave
int FieldCurrentLocal=0; //Store the value of FieldCurrent sent from the slave in
int FieldCurrentConstrained;
float volttest;
int voltmax=255;
int lightstep=0;
SoftwareSerial blueToothSerial(RxD, TxD); //blueToothSerial connected to RxD and TxD
SoftEasyTransfer ETin, ETout; //In/out object
Slave §
/*Program for Slave unit*/
#include <SoftEasyTransfer.h> //BT library
#include <SoftwareSerial.h> //SoftwareSerial library to create a Serial connection on any pin
#define RxD 6 //blueToothSerial receiving pin
#define TxD 7 //blueToothSerial transmitting pin
#define SSrVoltageOut 11 //Output pin connected to LP-filter due to PWM
#define RotorCurrent A0 //Input pin that reads the rotor current (in voltage)
int SSrVoltage=0; //Store the regulated value in SSrVoltage
int FieldCurrentLocal=0; //Store the value from input pin RotorCurrent
float volttest;
SoftwareSerial blueToothSerial(RxD, TxD); //blueToothSerial connected to RxD and TxD
SoftEasyTransfer ETin, ETout; //In/out object

Figure 24: Variable declaration of the two Arduino sketches.

Because Arduino is an open-source platform there are many great libraries written by skillful programmers available for download. Two libraries are included in

both sketches used in this thesis:

1. *SoftwareSerial* – This library makes it possible to create a serial connection on any pin.
2. *SoftEasyTransfer* – A Bluetooth library that makes it easy to send data back and forth between the two Arduinos. Note that the serial connection between Arduino and BTshield occurs on Pin6 and Pin7, therefore the jumpers on the UART multiplexer must be set accordingly, see Figure 25.

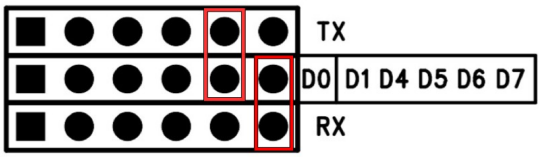


Figure 25: Jumper position of the UART multiplexer during operation.

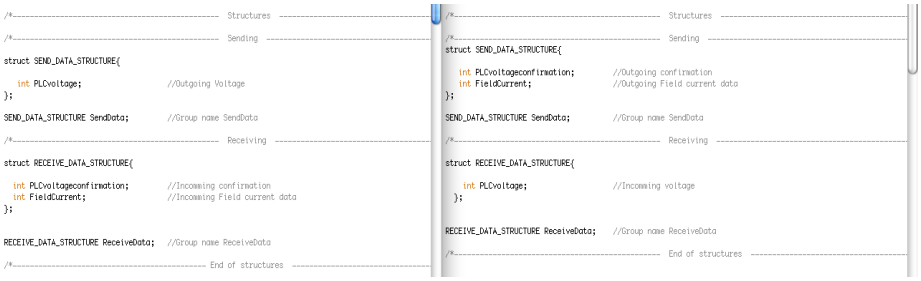


Figure 26: Structures inside the two sketches.

A transmitting structure and a receiving structure exist in both sketches, they contain the variables that are transferred by the Bluetooth connection. All data in the masters transmitting structure is transferred to the slaves receiving structure at a baud rate of 9600 and vice versa. This means that when a transmitting variable is changed, it almost instantaneously gets registered in the receiving structure of the interacting Arduino.

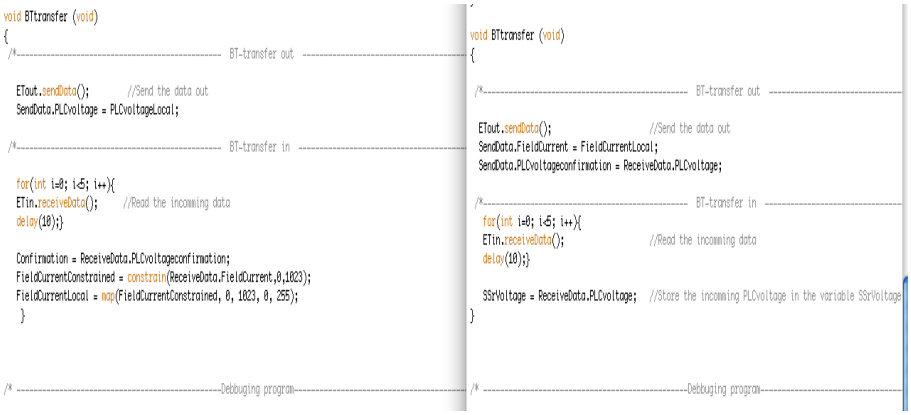


Figure 27: Data being assigned to structure variables.

sendData() and receiveData() are functions incorporated in the SoftEasyTransfer library. Each time they are called upon, data either gets pushed, out or received. BTtransfer, loops inside the sketch assigning the incoming and outgoing data to its internal- or structure-associated variable.
Other parts of the Arduino sketches can be seen in the Arduino appendix and are considered self explanatory.

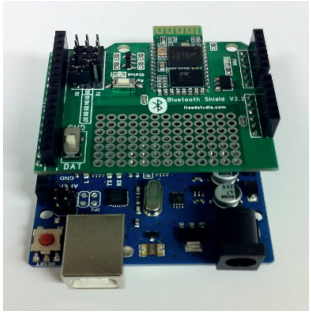


Figure 28: Arduino and the stackble BTshield mounted together.

5.3 Custom made shields

Two custom made stackable shields, one for the stationary Arduino and one for the Arduino on the rotating shaft had to be constructed to make them compatible with their connected devices.
Low-pass filters, transistor inverters and a differential amplifying circuit are all features of the two custom shields.

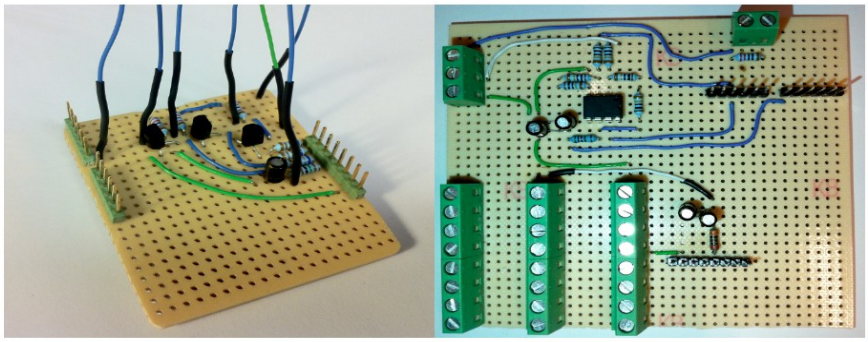


Figure 29: The stationary custom made shield (left) and the rotating custom made shield (right).

5.3.1 Low-pass filter

An analog output on the Arduino Uno delivers a PWM signal that has to be converted to an analog DC signal in order to be compatible with the different devices used throughout this thesis.
A low-pass filter will function as a D/A converter that transforms the PWM output to a DC voltage that is proportional to the duty cycle.

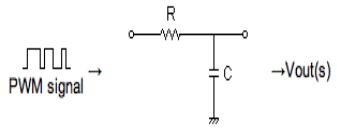


Figure 30: D/A conversion with a low-pass filter [23].

5.3.2 Transistor inverter

The Arduino inside the control cabinet sends the status of its power, connection and other information to the PLCs digital inputs. The transistor inverters are used because the PLC demand a higher voltage than the Arduino is capable of

delivering. Whenever the Arduino supplies a voltage on the input of the transistor inverter the output signal is equal to 0V, otherwise it's set to +Vs. A cable from the L+ terminal on the PLC delivers a Vs of 24V to all inverter circuits.

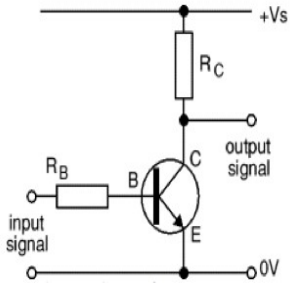


Figure 31: A transistor inverter [24].

5.3.3 Differential amplifying circuit

The rotating Arduinos custom shield includes a differential amplifying circuit that amplifies the current transducers output signal by a factor of two. This is done to enhance the measuring resolution since the transducers output only figures in the interval between 2.5V and 5V. The transducers output is fed to V_1 while a constant 5V is supplied to V_2 from the Arduino. The difference between the two voltages times the factor two, takes full advantage of the Arduinos capabilities, see equation (11).

$$R_1 = 100\text{k}\,,\ R_2 = 100\text{k}\,,\ R_f = 200\text{k}\,,\ R_g = 200\text{k}\,,\ V_2 = 5\text{V}$$

$$V_{out} = V_2 \frac{R_g(R_f + R_1)}{R_1(R_g + R_2)} - V_1 \frac{R_f}{R1} = 2(5 - V_1) \qquad (11)$$

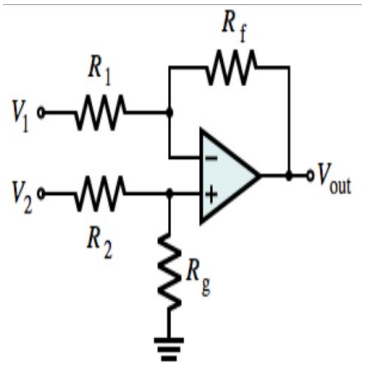


Figure 32: The differential amplifying circuitry used on the custom made shield belonging to the rotational Arduino [27].

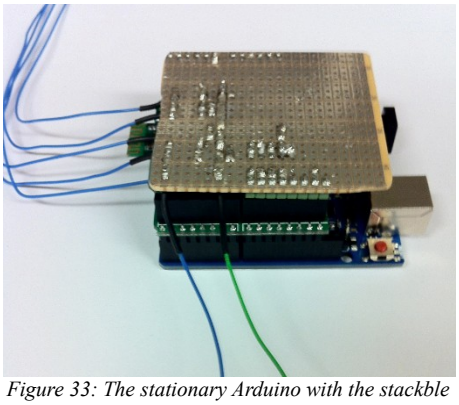


Figure 33: The stationary Arduino with the stackble BTshield and the stationary custom made shield mounted on top of it.

5.4 Programming the PLC

The concept of the PLC program is easiest understood if thought of as the workflow of the user interface. Figure 34 displays the workflow when operating the AVR system from the control panel or any other java-supported web-browser.

Steps:

1. The startup position of the project will let the user decide which one of the two exciters, static or rotating that the excitation system should use.
2. Once the exciter is chosen, the option to run the system manually or automatically is presented. In the background, parameters belonging to previously stored controllers are loaded from the SD card into a temporary array in the program.
3. Manual control: Here the user can manually adjust the control voltage fed to the SSRs, thereby controlling the amount of field current applied to the rotor winding.
4. The first step on the path of automatic control is choosing which controller to use. The user will choose between a P-, PI-, PD- and PID-controller. The parameters that were previously gathered in step 2 are now called upon to enable a quick-run option.
5. If the user decides to create new parameters for a controller then a numerical visualization is provided for manual configuration. New parameters can be stored on the SD card or used as temporary settings during the active cycle.
6. Automatic operation: Provides manual start and stop functionality to the AVR as well as a graphical plot of the actual- and reference-value of the controller. The program returns to step 1 once an exit is made and the cycle has ended.

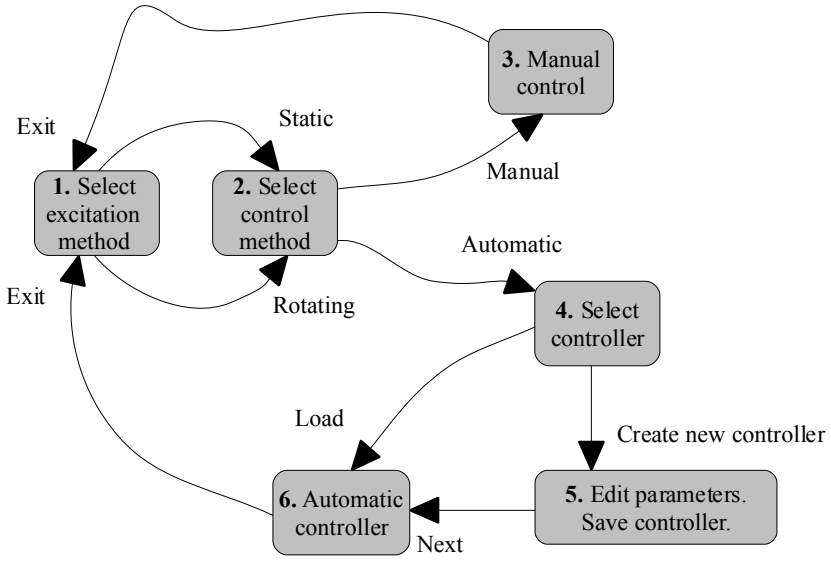


Figure 34: Operational workflow of the user interface.

5.4.1 POUs

16 different Program Organization Units completes the PLC project of the AVR.

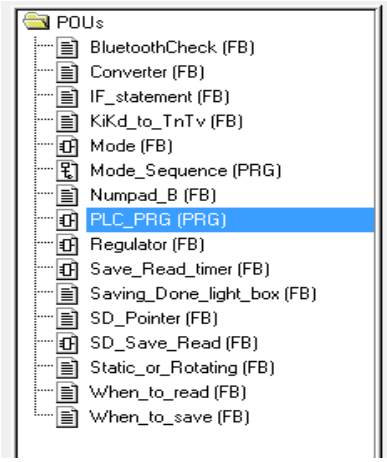


Figure 35: Screenshot of the different POUs created with CoDeSys.

PLC_PRG

This is the main program written in CFC and it is here all the other programs, functions and function blocks joins together and get interconnected. CFC produces a great graphical overview of the entire system making it easy to follow the values of the PLCs I/Os as well as the paths of the internal signals during operation or simulation.

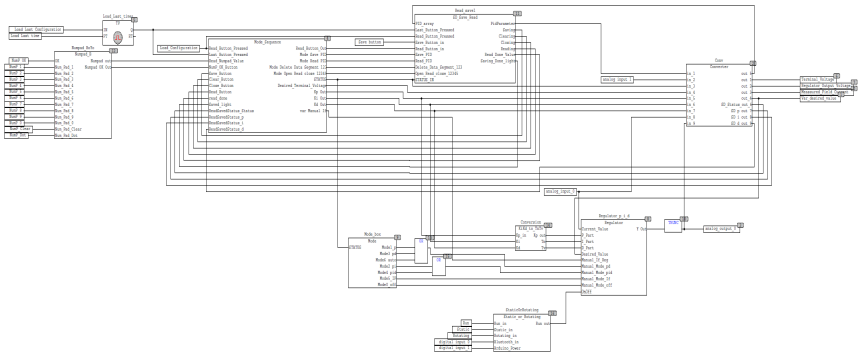


Figure 36: PLC_PRG shows the interconnection between the different parts of the PLC project.

Mode_Sequence

Written in SFC, Mode_Sequence is the program that keeps track of the projects position, see Figure 37. The different steps are mainly occupied by the controllers. Each transitions (mode0 to mode12) correspond to their own STATUS value. For example, if the pi-controller is selected from the control panel, STATUS will be assigned the value 2 which opens the transition Mode2, gaining the user access to its content after the system has finished reading the SD

card. Mode0 and Mode7 to Mode12 all open when STATUS equals zero, which occurs at every startup and every exit. After an exit is performed the project position gets pushed to Init-step before getting moved onwards to the off-step, where it resets important values and reads the SD card.

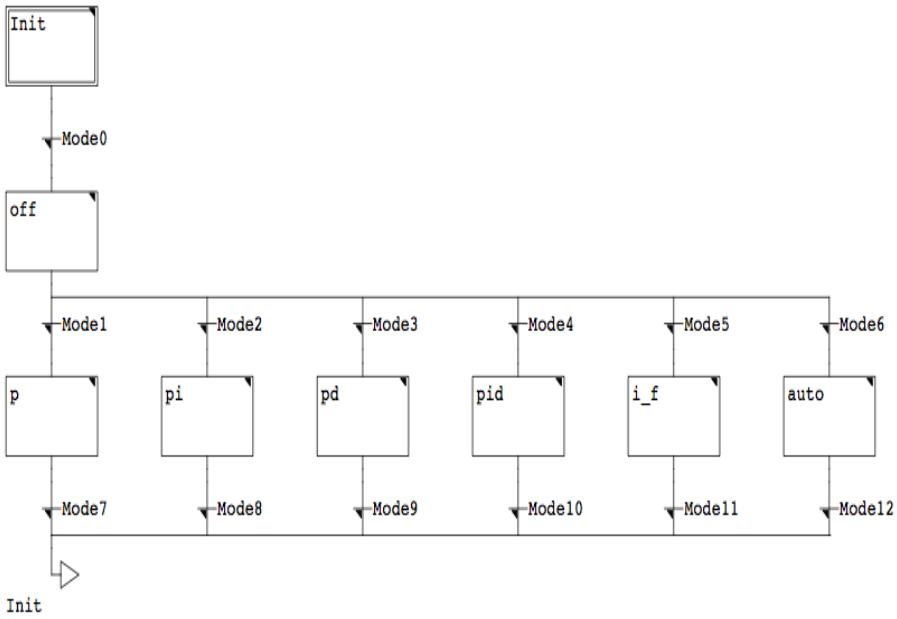


Figure 37: Each step of the Mode_Sequence POU holds the information needed for its corresponding position.

Actions assigned to steps:

- Init(ST)
  Checks if the user have pressed Back or Exit to determine if the system must read the SD card or not.

- off(ST)
  Resets controller parameters and other internal variables. Reads the SD card if the result of the Init-step activates the action.

- p(ST), pi(ST), pd(ST), pid(ST)
  A controller is activated after the STATUS value corresponding to its step is set. Then a user can load the saved parameters from the SD card or manually enter new values by commanding the Numpad_B POU. If the user opts to save a new value, then a command will be sent to the SD_Save_Read POU, overwriting the old one.

- i_f(ST)
  Can call the Numpad_B POU and lets the user manually adjust the field current.

- auto(ST)
  Not in use.

Numpad_B

This POU written in ST was created so that it would be possible to manually use the control panel to change and edit different parameters of the AVR. It is directly connected to the Numpad visualization, where every button corresponds to a part in the code. The value of the button being pushed gets added to a specific part of an array that gets merged into a string before being converted to a real-type value after the OK button is pushed. Timers from the CoDeSys library are used as debounce functions for the buttons to avoid typos.

Mode

This is a function block written in FBD that checks the value of the variable STATUS to activate the correct mode of the system. The output of this POU decides what parts of the Regulator POU that should be active.

Static_or_Rotating

A function block written in ST that acts as a safety measure to the system, making it impossible to run the AVR if the brushless exciter and the static exciter both are active.

KiKd_to_TnTv

The controllers inside the CoDeSys library expects the integral action time, T_n and the derivative action time, T_v as inputs instead of k_i and k_d. This function block written in ST calculates the action times from the relations presented in equations (6) and (9), using the parameters k_i and k_d submitted by the user.

Regulator

Regulator is a function block written in CFC that handles the actual regulating of the AVR. The two function blocks (PD and PID) are provided by the CoDeSys library, the combination enables the AVR to produce all controllers necessary, including the P- and PI-controller. The actual value needed by both controllers comes from the variable Current_Value that is supplied by an integer corresponding to the terminal voltage measured by the voltage transducer. Both the regulated outputs are constrained by the controllers so that they don't rise above the 5V maximum limit of the Arduino receiving pin.
Aside from the two controller function blocks there is the IF_statement POU that is activated when manual operation is selected.
Inputs coming from the Mode POU decides which one out of the three units that is to be activated, while the other two are kept offline.
The BluetoothCheck POU is another safety measure, installed to ensure that the produced regulated output don't fluctuate if the Bluetooth connection were to be lost.

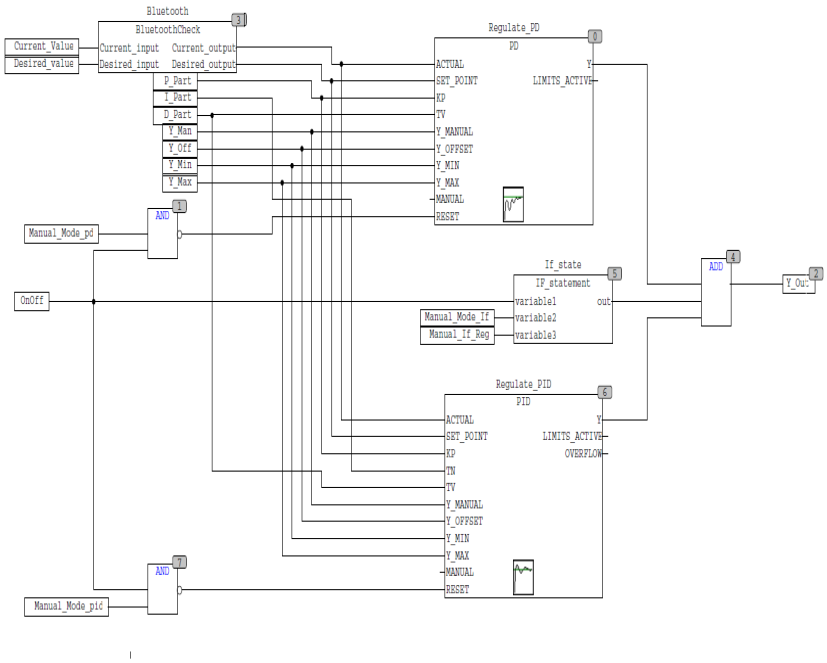


Figure 38: The Regulator POU contains all the different controllers used for automatic control as well as the IF_statement POU which handles the manual operation.

IF_statement

As previously stated the function block IF_statement is utilized during manual operation. It consists of a few rows of code written in ST that passes on the input Manual_If_Reg from the Regulator POU to its output when activated. The variable Manual_If_Reg is directly connected to the Manual visualization and adjustable up to 5V with the help of the created numerical keyboard.

BluetoothCheck

The PLCs digital input0 receives the status of the Bluetooth connection from the master Arduino. While the connection is functioning the AVR will run as normal, but if the connection is lost, then the two outputs of the BluetoothCheck POU are assigned the same value in order to create $e(t) = 0$ according to equation (2).

SD_Save_Read

This is a function block written in CFC that lets the user save and load controller parameters from the control panel. SD_WRITE and SD_READ are both function blocks included in the CoDeSys library that connect directly to a SD card inserted into the PLC. The elements that are to be stored on the SD card can not be of a real data type, this means that the controller parameters have to be configured before being saved. A thirteen element long array is stored on the SD card with the altered parameters of all the optional controllers provided by the Converter POU, see Table 5. This array gets overwritten every time a save is made, limiting the data on the SD card to only one file.

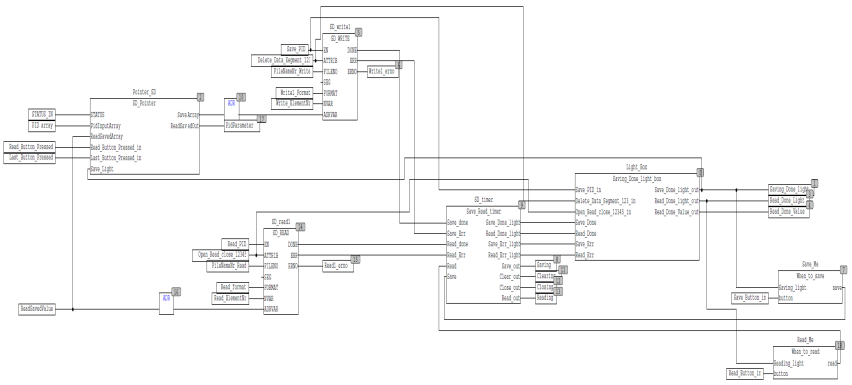


Figure 39: SD_Save_Read is used to save and load controller parameters.

Status	P-Controller			PI-Controller			PD-Controller			PID-Controller		
Last	k_p	0	0	k_p	k_i	0	k_p	0	k_d	k_p	k_i	k_d

Table 5: The array stored on the SD card containing the status value belonging to the last saved controller, followed by the altered parameters of the different controllers.

SD_Pointer

This POU points the active parameters to their correct place in the array when saving. The first element of the array belongs to the status value of the controller that was last saved, by storing it the user may load the last saved controller quickly from the control panel.

Save_Read_Timer

A certain sequence of commands must be sent to SD_Write and SD_Read every time they are being used. Save_Read_Timer utilizes the CoDeSys timers to ensure that the sequences follow the correct order.

1. The SD card must be erased before saving.
2. The file must be closed after a load-process is completed.

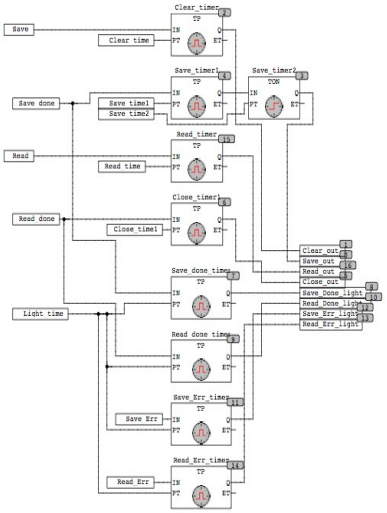


Figure 40: Save_Read_Timer is a function block written in CFC that sets the sequential order of the inputs belonging to SD-Read and SD-write.

Saving_Done_light_box

The main purpose of the Saving_Done_light_box function block is to provide visual aid of the save and load process.

When_to_save

A safety function block to prevent the system from trying to save while a saving process is ongoing.

When_to_read

A safety function block to prevent the system from trying to read while a reading process is ongoing.

Converter

The converter fills an important role in the project.

- Provides the SD_Save_Read with compatible values to be saved.
- Converts the values back to controller parameters when loading.
- Calculates voltage and current equivalents of the PLCs inputs received from the voltage transducer and the Arduino. The calculated values are used in visualizations and for debugging purposes.

5.4.2 Visualizations

The visualizations created in CoDeSys form the operating webpage interface of the PLCs web server application. Buttons and sliders creates an HMI where the user can trigger different actions of the AVR and edit values associated with the Regulator POU.

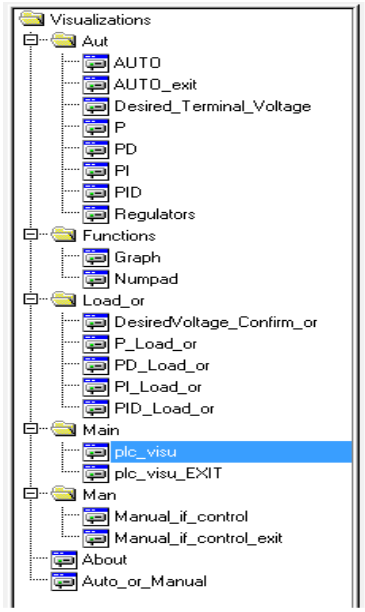


Figure 41: The name plc_visu is pre-assigned to the startup window of the user interface (like PLC_PRG).

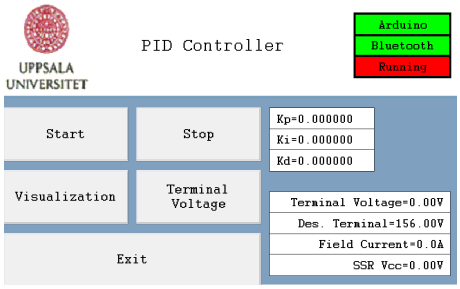


Figure 42: Operational window during automatic control.

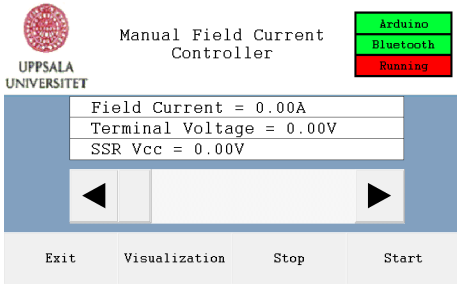


Figure 43: Operational window during manual control.

5.5 The control cabinet

This thesis was performed with the aim of also building the complete AVR system. Once the programable devices and the Bluetooth connection had been finalized and established, all parts of the AVR that were considered stationary got put into a control cabinet. The cabinet has the control panel mounted on its exterior door to enable easy operational access as seen in Figure 45. Figure 44 displays the different stationary devices used by the control cabinet, they are all illustrated in Figure 1 with their purpose presented in the System overview section of the Introduction chapter. A more detailed description of the functionalities of some of them can be found in the Hardware chapter.

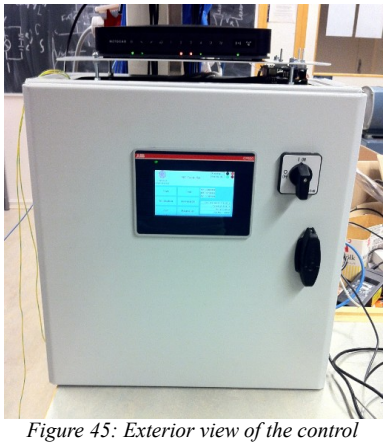


Figure 45: Exterior view of the control cabinet.

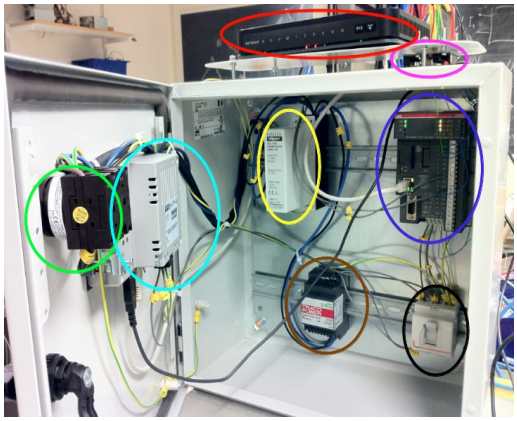


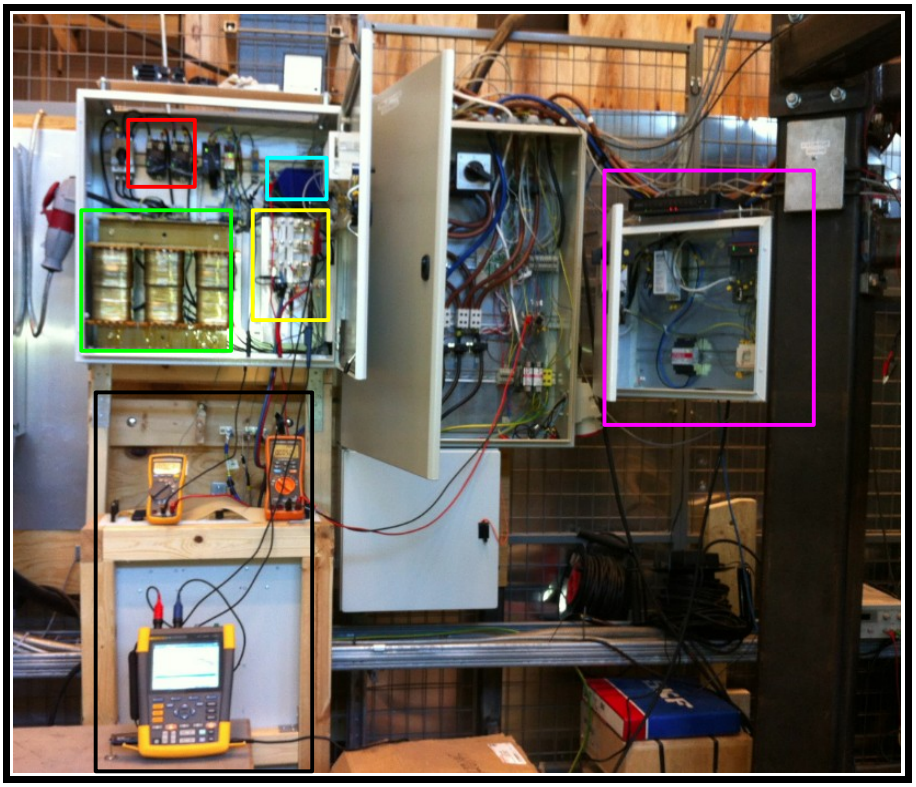
Figure 44: Interior view of the control cabinet. The different components of the stationary AVR are highlighted in different colors.

	Device	Component walkthrough	Hardware description
	On/off switch	20	3.8
	Control panel	15	3.2
	Router	16	3.8
	Arduino+BTshield+Custom made shield	5+4+6	3.3 +3.4 +5.3
	Voltage transducer	12	3.5
	PLC	14	3.1
	AC/DC converter	19	3.8
	Safety switch	21	3.8

Table 6: The devices belonging to the control cabinet and references to them within this thesis.

5.6 The excitation system

The completed construction of the AVR system is implemented on the already existing experimental setup, with the control cabinet welded to the bearing structure of the SGs stator. Test runs are performed by using a static exciter that is controllable by the rotational parts of the AVR. The exciter consists of a step-down transformer with a 7.2:1 ratio, meaning that a primary voltage of 400VAC results in a 55.5VAC on its secondary winding. Two of the three phases leading out from the transformer and in to the three-phase diode bridge rectifier have controllable SSRs connected to them, while the third one is connected directly. The control signal is received through the wireless Bluetooth connection and fed to the SSRs by the rotational Arduino and its associated shields that are placed in the blue box above the rectifying circuitry as seen in the figures bellow.



	Transformer
	SSRs
	Rotating Arduino + BTshield + custom made shield
	Three-phase diode bridge rectifier
	Measuring equipment
	Control cabinet

Figure 46: The experimental setup.

6 Results

This chapter provides a performance review of the constructed AVR system, when connected to the static exciter. The different controllers obtained by the Ziegler-Nichols tuning rule are tested and problems relating to the SSRs and the Bluetooth connection are addressed.

6.1 Applying the Ziegler-Nichols tuning rule

The Ziegler-Nichols turning rule is used to find suitable parameter values for the PLC controllers when connected to the SG Svante. The method is performed according to the instructions presented in section 2.4 of the Theory chapter, using a computer to wirelessly control the system. The visualization tool of the HMI provides the information needed to compute the first step of acquiring the s_u and p_u values, see Figure 47.

- $s_u = 0.7$
- $p_u = 2.3$

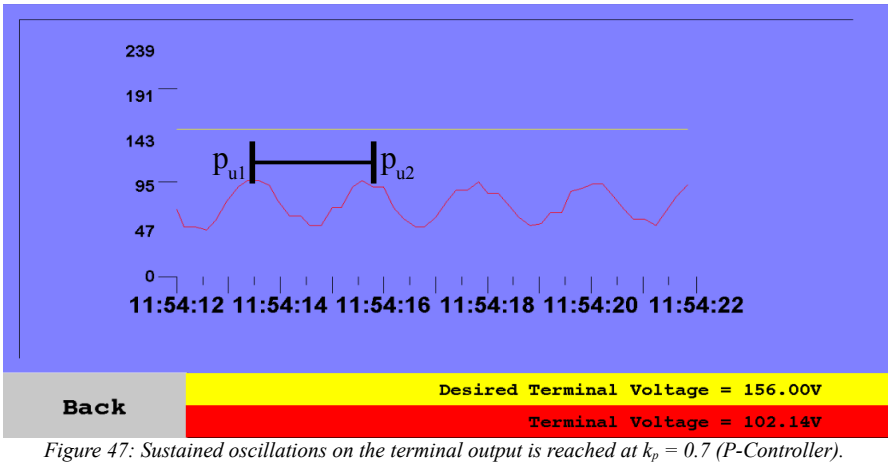


Figure 47: Sustained oscillations on the terminal output is reached at $k_p = 0.7$ (P-Controller).

Now that the ultimate sensitivity, s_u, and the ultimate period, p_u, have been acquired, the parameters belonging to the different controllers listed in Table 3 and 4 are calculated and presented in Table 7.

	k_p	k_i	k_d
P-Controller	0.350	-	-
PI-Controller	0.318	0.166	-
PID-Controller	0.420	0.365	0.121
PID with no overshoot	0.140	0.122	0.107
PID with some overshoot	0.231	0.201	0.177
PID with Pessen integral rule	0.490	1.750	0.169

Table 7: Calculated parameter values for the different controllers provided by the tuning rules.

6.2 Controller performances - step responses

A step response of the finalized closed-loop system for each of the different controllers calculated in Table 7 is obtained by monitoring the terminal voltage during a stop/start of the AVR. The off/on procedure sets the regulating process in motion by creating a step of the reference value going from 0V to 156V.
The PID-Controller determined by the Ziegler-Nichols tuning rule complies the most with the requirements wanted from a solid step response with regards to settling time, rise time and overshoot. Furthest away from a desired output is the PID-Controller belonging to the Pessen integral rule, which produces an unstable oscillating terminal output on the SG. Screenshots of the different step responses (except P-Controller) are collected from the visualization tool of the HMI and displayed in the figures bellow.

PI-Controller

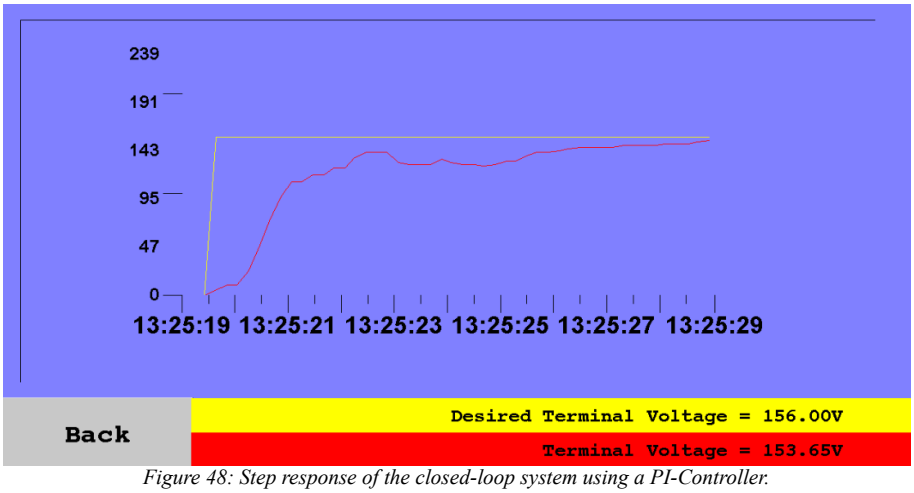


Figure 48: Step response of the closed-loop system using a PI-Controller.

PID-Controller

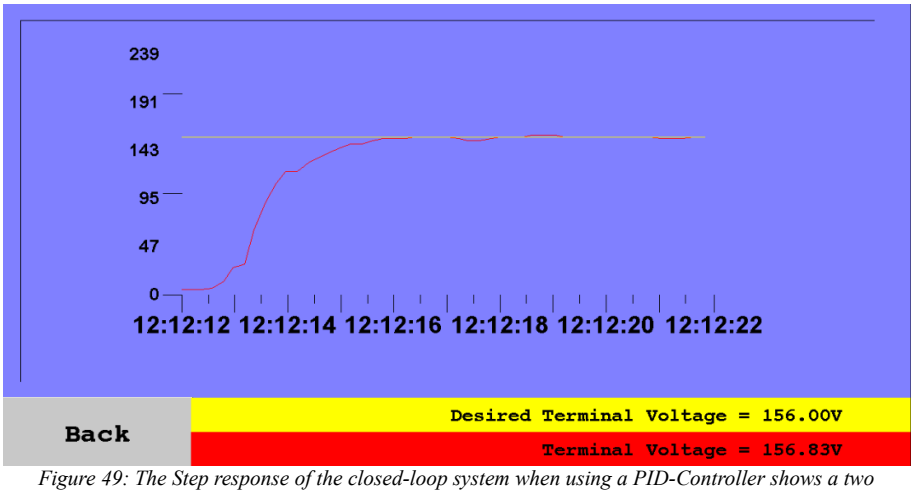


Figure 49: The Step response of the closed-loop system when using a PID-Controller shows a two seconds long settling time.

PID with no overshoot

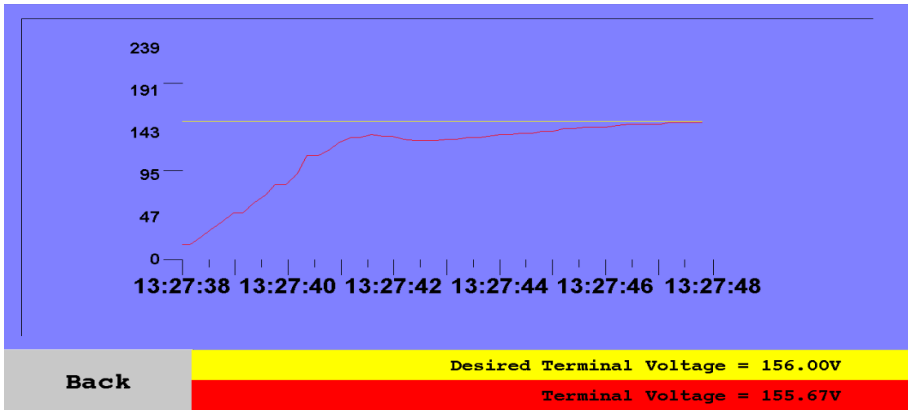


Figure 50: Step response of the closed-loop system using a PID with no overshoot.

PID with some overshoot

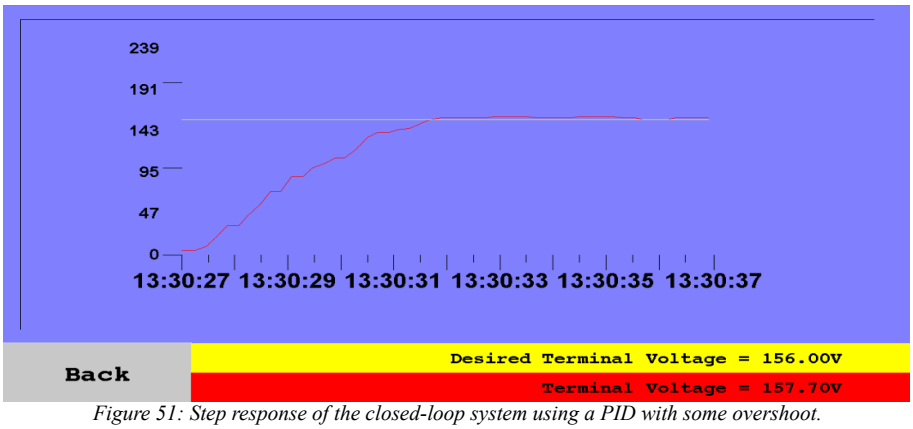


Figure 51: Step response of the closed-loop system using a PID with some overshoot.

Pessen integral rule

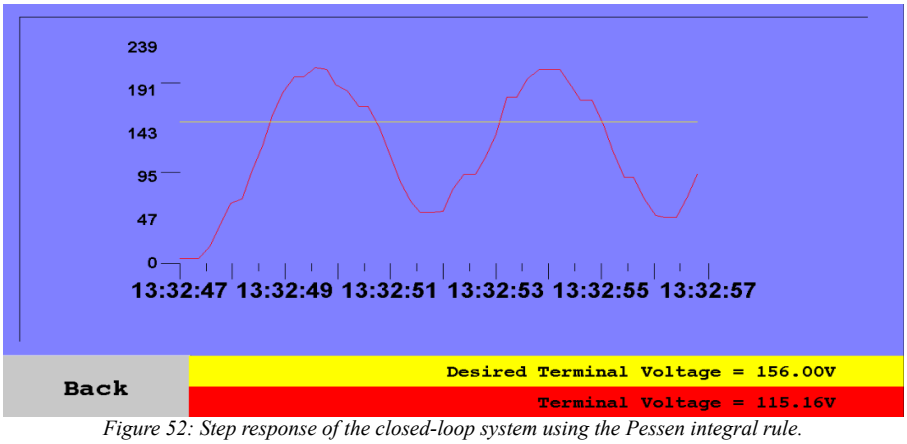


Figure 52: Step response of the closed-loop system using the Pessen integral rule.

6.3 Manual operation

Figure 53 displays the behavior of the terminal voltage when a 3.05V control voltage is supplied to the SSRs during manual operation. Because no controller is applied, the settling time of the curve represents a time delay mainly caused by the rotor inductance.

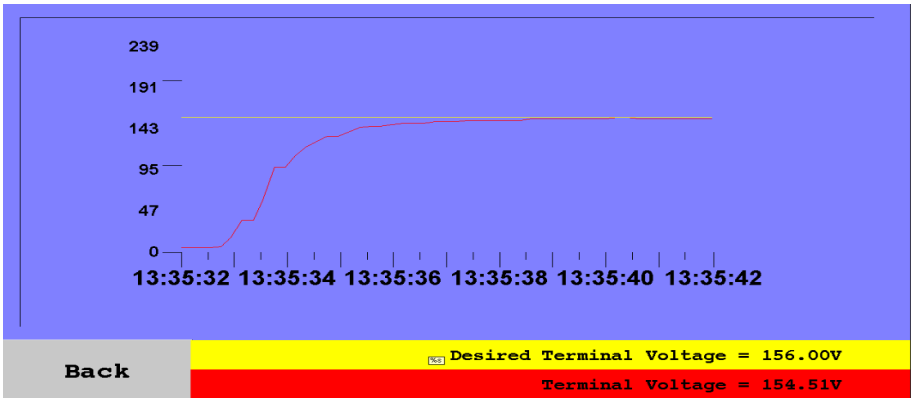


Figure 53: Terminal voltage during manual operation @ 3.05V.

6.4 SSRs

Problems related to the two SSRs became clear during manual operation. An increased control voltage should result in an increased field current, but as seen in the two figures bellow the behavior is exactly the opposite.

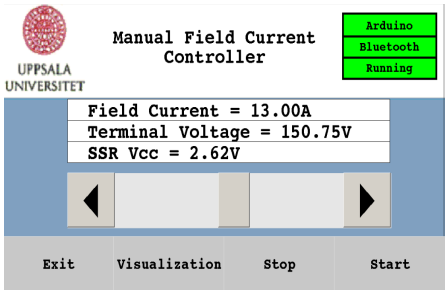


Figure 55: 150.75V at 2.62V control voltage during manual operation.

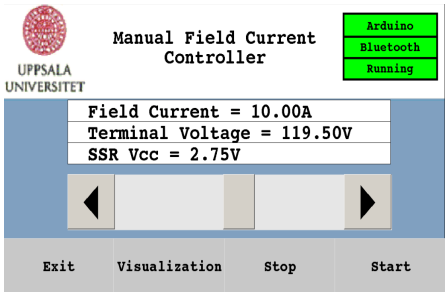


Figure 54: 119.5V at 2.75V control voltage during manual operation.

6.5 The Bluetooth connection

The binding of the Bluetooth addresses made the devices connect to each other fairly rapidly (seconds) after a power-up. The data stream that follows runs smoothly and in the designated speed set by the baud rate. However, connection failures between the two BTshields have occurred during operation, lasting for less than a second and seemingly occurring at random. When or if a failure occurs the rotating Arduino will keep a constant control signal fed to the SSRs at the level in which connection was lost. The protective code programmed in the

PLC for this scenario (BluetoothCheck) will also freeze the value of the control signal which prevents the rotating Arduino from dropping it down to zero once a connection is re-established. The safety measures put in to place have proven themselves to be sufficient enough to ensure a non-fluctuating terminal voltage during a Bluetooth connection failure.

7 Conclusions

The work done during the course of this thesis have reached far beyond what had been initially intended. Even so, it has been greatly rewarding to be able to build the entire control system from start to finish. Even though the AVR constructed preforms all tasks that it was required to do, there is still plenty of room for improvements to be made, see the chapter Future work.
Focus has mainly been directed towards the programming aspect of the AVR, thus other parts haven't gotten the same coverage within this paper. The programming code that have been written is the results of several hours of trial and error, compiling and simulation. The final version of the written code have been tested numerous times without any errors, therefore, future work should focus on the AVRs other parts.

8 Future work

Listed bellow are a couple of suggestions on how to further develop the AVR system.

SSR

The SSRs used in the AVR system didn't work as intended when connected on every phase, therefore a fixing solution should be considered.

When three SSRs were placed on eace of the three phases that connects to the rectifying circuitry they kept blocking each other so that none of them were capable of conducting. The zero-crossing SSRs became operational after one of them was removed from its belonging phase. The removal of the SSR affected the quality of the DC voltage leading out from the bridge rectifier. This method proved itself to be inadequate during manual operation as seen in section 6.4 of the Results chapter. Someone must either find a way to make the SSRs work as intended or come up with an alternative solution to control the voltage level. One idea is to not have any SSRs before the bridge rectifier and only use one DC-SSR to control the DC voltage instead.

Controller

The parameters produced by the Ziegler-Nichols tuning rule provide a solid foundation to start with. However, adjustments should be made in order to obtain even better results. Another option is to mathematically calculate the transfer-function of the entire system and use it to find the optimal controller.

PLC - Arduino communication

The communication that takes place between the PLC and the stationary Arduino is done by measuring DC voltages. The PLC and the Arduino interpreters their received input voltages as a data type so that it may be used for its intended purpose.

It would be preferable to use a serial communicating link between the two devices, which would eliminate the need for the custom made shield on the Arduino. It is possible to connect the USB port of the Arduino to the serial communication port on the PLC, just like when connecting to a computer. Not only would data be sent faster and more accurate but the 10 bit resolution of the analog input on the rotating Arduino would be used to its fullest.

Bluetooth communication

The Bluetooth connection is fully functional but not optimized in any way. An in-depth study on how Bluetooth communicating devices work, might provide an explanation to why the two BTshields tend to loose connection and how it can be prevented.

Code

Even though the code written for the PLC and the two Arduinos seem to work flawlessly. It's still possible to optimize it to conserve memory and to make it run smoother.

Additional functionality

With the structure of the PLC program in place, it shouldn't be a problem to add new features that could be accessible from the HMI interface. A new controller that can maintain a constant $\cos(\varphi)$ should be implement to the PLC project, as well as more use of the PLCs inputs to further support the regulating process.

9 References

[1] Power Generation Handbook: Selection, Applications, Operation and Maintenance; Philip Kiameh; McGraw-Hill Professional; 1 edition (Aug 28 2002)

[2] The Electric Generators Handbook: Synchronous Generators; Ian Boldea; Taylor & Francis 12 dec 2010

[3] Handbook of Electrical Power System Dynamics: Modeling, Stability, and Control; Mircea Eremia and Mohammad Shahidehpour; Wiley-IEEE Press April 2013

[4] Power System Stability and Control: Prabha Kundur; McGraw-Hill Professional (January 1 1994)

[5] Feedback Systems: An Introduction for Scientists and Engineers Karl Johan Åström; Richard M. Murray; Version v2.10b (February 22, 2009)

[6] Optimum Settings for Automatic Controllers: J.G. ZIEGLER and N. B. NICHOLS; ROCHESTER, N. Y.

[7] http://www.arduino.cc/ 2013-06-10

[8] User Manual for PLC Programming with CoDeSys 2.3; Version 2.0; 3S - Smart Software Solutions GmbH

[9] Getting Started with Arduino; Massimo Banzzi; First Edition 2008

[10]http://www05.abb.com/global/scot/scot209.nsf/veritydisplay/a9d2e5d590304dc7c125799800336877/$file/1sbc125003c0204-automation%20products_new-br.pdf 2013-06-10

[11] Measurement and modelling of of unbalanced magnetic pull in hydropower generators; Mattias Wallin; Thesis, Uppsala University

[12]http://www05.abb.com/global/scot/scot209.nsf/veritydisplay/a9ce342e2d3cac8dc125790a0035e777/$file/2CDC159008M0201.pdf 2013-06-10.

[13] http://www.bluetooth.com/Pages/Bluetooth-Home.aspx 2013-06-10.

[14] http://www.developer.nokia.com/Community/Wiki/Bluetooth_Protocol 2013-06-10

[15]http://www.abb.com/product/seitp329/be2c616b9f2fc3e2c1256ff00047a8f2.aspx?country=00&tabKey=2&cid=9AAC100147&gid=ABB1TNE968900R1110 2013-06-12

[16] http://www.abb.com/product/seitp329/6b9c7e5f00fb4961c125790600299bbc.aspx?country=00&tabKey=2&cid=9AAC124505&gid=ABB1SAP520100R0001 2013-06-12

[17] HC Serial Bluetooth Products User Instructional Manual http://www.mcu-turkey.com/wp-content/uploads/2013/01/HC-Serial-Bluetooth-Products-201104.pdf 2013-06-12

[18] http://imall.iteadstudio.com/im120417010.html 2013-06-12

[19] http://www.elecfreaks.com/store/images/product_images/Wireless/bluetooth/bluetooth-HC05-02.jpg 2013-06-12

[20] http://www.tillquist.com/eng/images/stories/datablad/iu420-datasheet.pdf 2013-06-12

[21] http://www.seekic.com/uploadfile/icdatasearch/CSLA/10310858383296141.jpg 2013-06-12

[22] http://www.crydom.com/en/Tech/Newsletters/Solid%20Statements%20-%20SSRs%20switching%20types.pdf 2013-06-12

[23] http://sim.okawa-denshi.jp/en/CRlowkeisan.htm 2013-0612

[24] http://electronicsclub.info/images/trinvert.gif 2013-06-12

[25] Ziegler-Nichols' Closed-Loop Method; Finn Haugen; TechTeach 17. July 2010

[26] The father of invention: Dick Morley looks back on the 40th anniversary of the PLC; Manufacturing Automation , written by Alison Dunne 12 September 2008.

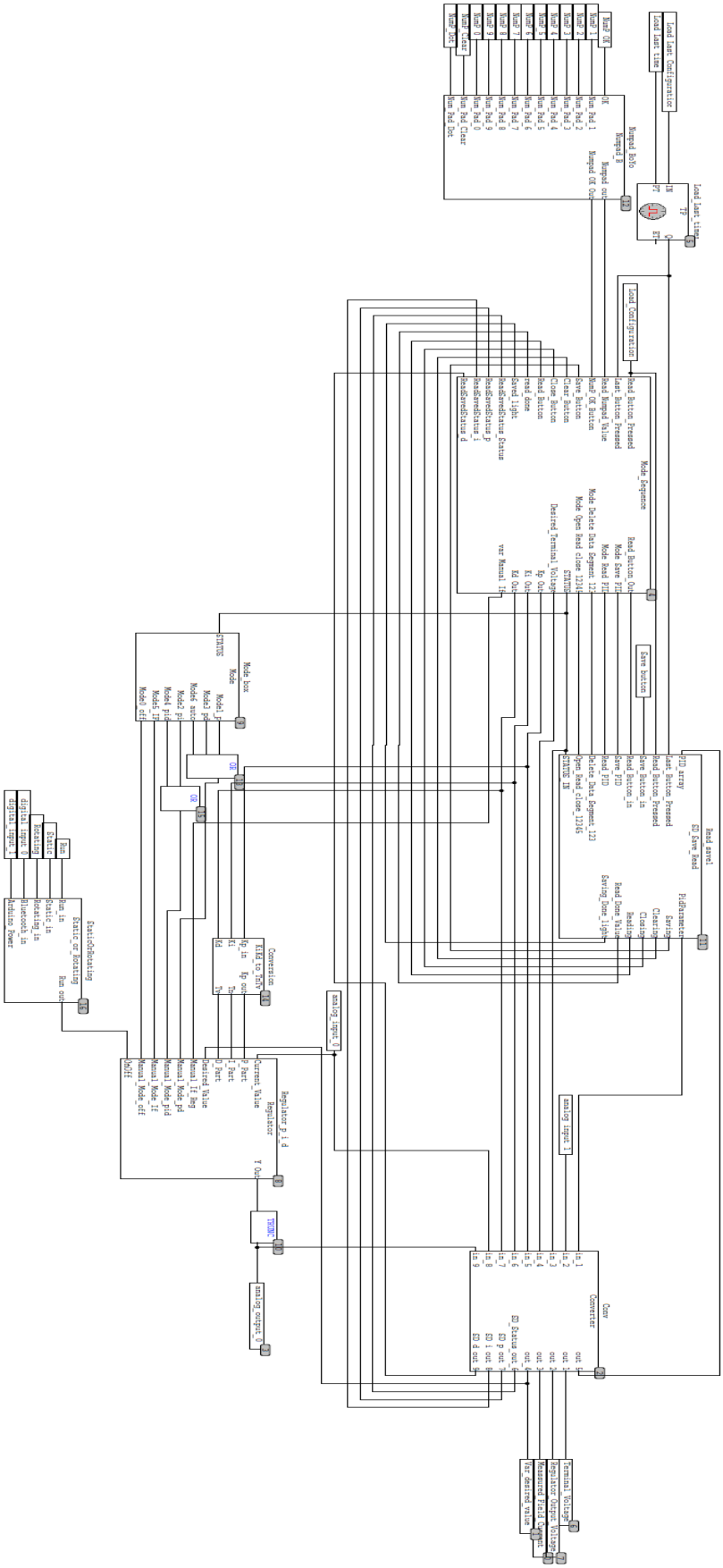
[27] https://upload.wikimedia.org/wikipedia/commons/a/a2/Op-Amp_Differential_Amplifier.svg 2013-06-13.

10 Appendices

PLC code

PLC_PRG

```
0001 (*Main program*)
0002
0003 PROGRAM PLC_PRG
0004 VAR
0005     Conv: Converter;
0006     Regulator_p_i_d: Regulator;
0007     Mode_box: Mode;
0008
0009     Terminal_Voltage: REAL;
0010
0011     Regulator_Output_Voltage: REAL;
0012     Var_desired_value: REAL;
0013
0014     Run: BOOL;
0015
0016     Meassured_Field_Current: REAL;
0017     Read_save1: SD_Save_Read;
0018
0019     Numpad_BoYo: Numpad_B;
0020     NumP_1: BOOL;
0021     NumP_2: BOOL;
0022     NumP_3: BOOL;
0023     NumP_4: BOOL;
0024     NumP_5: BOOL;
0025     NumP_6: BOOL;
0026     NumP_7: BOOL;
0027     NumP_8: BOOL;
0028     NumP_9: BOOL;
0029     NumP_0: BOOL;
0030     NumP_Dot: BOOL;
0031     NumP_Clear: BOOL;
0032     NumP_OK: BOOL;
0033
0034     Load_Configuration: BOOL;
0035     Load_Last_Configuration: BOOL;
0036     Save_button: BOOL;
0037
0038     Load_Last_timer: TP;
0039     Load_Last_time: TIME:=T#500ms;
0040     Conversion: KiKd_to_TnTv;
0041     StaticOrRotating: Static_or_Rotating;
0042     Static: BOOL;
0043     Rotating: BOOL;
0044 END_VAR
0045
0046 VAR_INPUT
0047
0048 END_VAR
0049
```



Numpad_B

```
0001 FUNCTION_BLOCK Numpad_B
0002 VAR_INPUT
0003     OK:BOOL;
0004     Num_Pad_1: BOOL;
0005     Num_Pad_2: BOOL;
0006     Num_Pad_3: BOOL;
0007     Num_Pad_4: BOOL;
0008     Num_Pad_5: BOOL;
0009     Num_Pad_6: BOOL;
0010     Num_Pad_7: BOOL;
0011     Num_Pad_8: BOOL;
0012     Num_Pad_9: BOOL;
0013     Num_Pad_0: BOOL;
0014     Num_Pad_Clear: BOOL;
0015     Num_Pad_Dot: BOOL;
0016 END_VAR
0017
0018 VAR_OUTPUT
0019     Numpad_out: REAL; (*Sending numpad values to Mode_sequence*)
0020
0021     Numpad_OK_Out: BOOL;
0022 END_VAR
0023
0024 VAR
0025     Var_array: ARRAY [1..8] OF STRING;      (*Maximum of 8 values*)
0026     no_of_push: INT := 0;                   (*Keeps track of the input value position 1-8*)
0027     Numpad_Display: STRING;                 (*String value for the numpad display*)
0028
0029 (*TPInstX is a timer for debounce*)
0030     TPInst1: TP;
0031     Numpad_Debounce_1: BOOL;
0032     TPInst2: TP;
0033     Numpad_Debounce_2: BOOL;
0034     TPInst3: TP;
0035     Numpad_Debounce_3: BOOL;
0036     TPInst4: TP;
0037     Numpad_Debounce_4: BOOL;
0038     TPInst5: TP;
0039     Numpad_Debounce_5: BOOL;
0040     TPInst6: TP;
0041     Numpad_Debounce_6: BOOL;
0042     TPInst7: TP;
0043     Numpad_Debounce_7: BOOL;
0044     TPInst8: TP;
0045     Numpad_Debounce_8: BOOL;
0046     TPInst9: TP;
0047     Numpad_Debounce_9: BOOL;
0048     TPInst0: TP;
0049     Numpad_Debounce_0: BOOL;
0050     TPInstDot: TP;
0051     Numpad_Debounce_Dot: BOOL;
0052
0053     Con_12: STRING;
0054     Con_34: STRING;
0055     Con_56: STRING;
0056     Con_78: STRING;
0057
0058     Con_1234: STRING;
0059     Con_5678: STRING;
0060 END_VAR
```

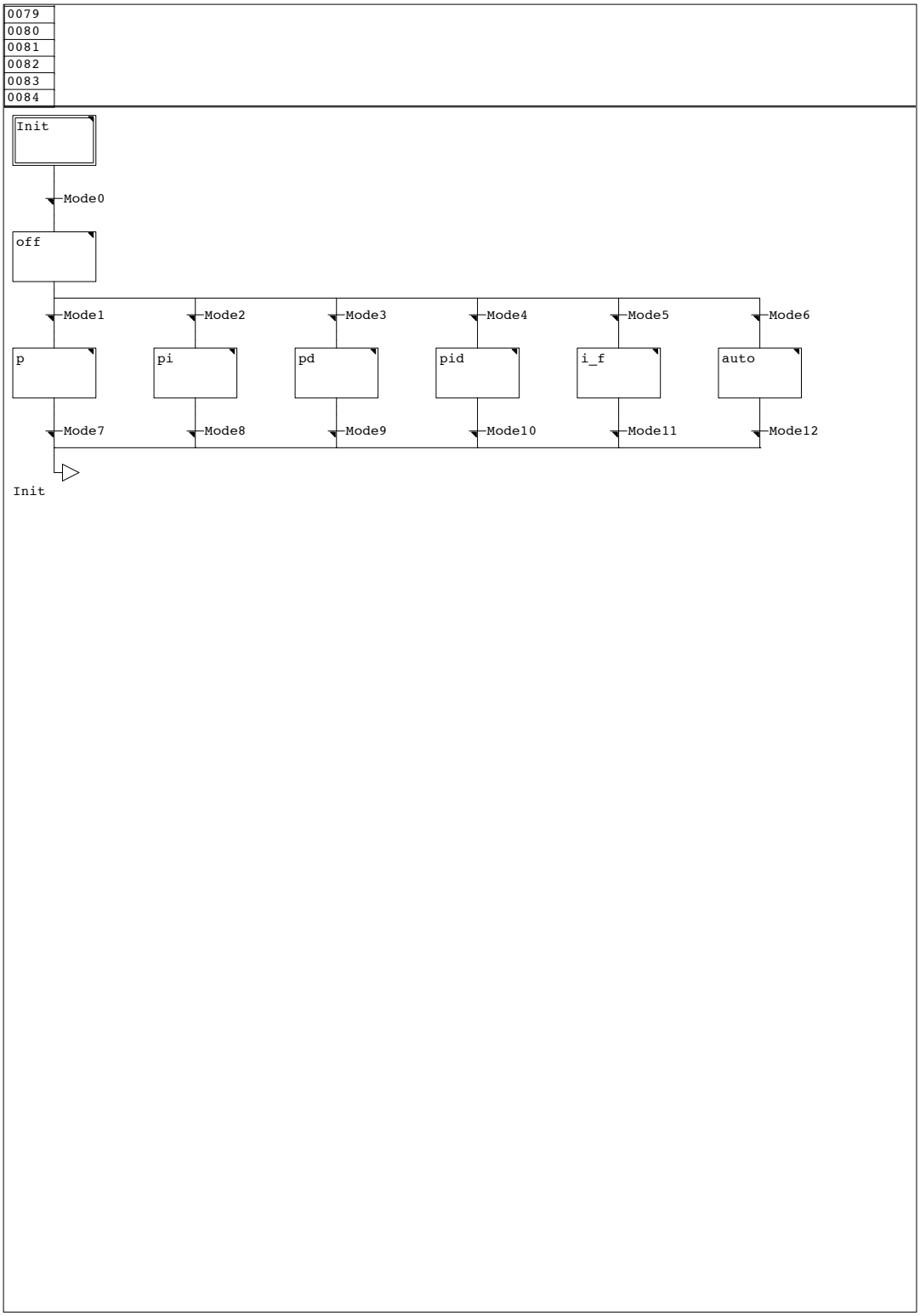
```
0001 Numpad_OK_Out:=OK;
0002 (*Clearing the Numpad values*)
0003 IF Num_Pad_Clear
0004 THEN
0005 no_of_push:=0;
0006 Var_array[1]:='';
0007 Var_array[2]:='';
0008 Var_array[3]:='';
0009 Var_array[4]:='';
0010 Var_array[5]:='';
0011 Var_array[6]:='';
0012 Var_array[7]:='';
0013 Var_array[8]:='';
0014 END_IF;
0015
0016 (*Debounce button function*)
0017 TPInst1(IN := Num_Pad_1, PT:= T#5ms);
0018 Numpad_Debounce_1 :=TPInst1.Q;
```

```
0019 (*Button 1 pressed*)
0020 IF Numpad_Debounce_1
0021 THEN
0022 no_of_push:=no_of_push+1;
0023 Var_array[no_of_push]:='1';
0024 END_IF;
0025
0026 (*Debounce button function*)
0027 TPInst2(IN := Num_Pad_2, PT:= T#5ms);
0028 Numpad_Debounce_2 :=TPInst2.Q;
0029 (*Button 2 pressed*)
0030 IF Numpad_Debounce_2
0031 THEN
0032 no_of_push:=no_of_push+1;
0033 Var_array[no_of_push]:='2';
0034 END_IF;
0035
0036 (*Debounce button function*)
0037 TPInst3(IN := Num_Pad_3, PT:= T#5ms);
0038 Numpad_Debounce_3 :=TPInst3.Q;
0039 (*Button 3 pressed*)
0040 IF Numpad_Debounce_3
0041 THEN
0042 no_of_push:=no_of_push+1;
0043 Var_array[no_of_push]:='3';
0044 END_IF;
0045
0046 (*Debounce button function*)
0047 TPInst4(IN := Num_Pad_4, PT:= T#5ms);
0048 Numpad_Debounce_4 :=TPInst4.Q;
0049 (*Button 4 pressed*)
0050 IF Numpad_Debounce_4
0051 THEN
0052 no_of_push:=no_of_push+1;
0053 Var_array[no_of_push]:='4';
0054 END_IF;
0055
0056 (*Debounce button function*)
0057 TPInst5(IN := Num_Pad_5, PT:= T#5ms);
0058 Numpad_Debounce_5 :=TPInst5.Q;
0059 (*Button 5 pressed*)
0060 IF Numpad_Debounce_5
0061 THEN
0062 no_of_push:=no_of_push+1;
0063 Var_array[no_of_push]:='5';
0064 END_IF;
0065
0066 (*Debounce button function*)
0067 TPInst6(IN := Num_Pad_6, PT:= T#5ms);
0068 Numpad_Debounce_6 :=TPInst6.Q;
0069 (*Button 6 pressed*)
0070 IF Numpad_Debounce_6
0071 THEN
0072 no_of_push:=no_of_push+1;
0073 Var_array[no_of_push]:='6';
0074 END_IF;
0075
0076 (*Debounce button function*)
0077 TPInst7(IN := Num_Pad_7, PT:= T#5ms);
0078 Numpad_Debounce_7 :=TPInst7.Q;
0079 (*Button 7 pressed*)
0080 IF Numpad_Debounce_7
0081 THEN
0082 no_of_push:=no_of_push+1;
0083 Var_array[no_of_push]:='7';
0084 END_IF;
0085
0086 (*Debounce button function*)
0087 TPInst8(IN := Num_Pad_8, PT:= T#5ms);
0088 Numpad_Debounce_8 :=TPInst8.Q;
0089 (*Button 8 pressed*)
0090 IF Numpad_Debounce_8
0091 THEN
0092 no_of_push:=no_of_push+1;
0093 Var_array[no_of_push]:='8';
0094 END_IF;
0095
0096 (*Debounce button function*)
```

```
0097  TPInst9(IN := Num_Pad_9, PT:= T#5ms);
0098  Numpad_Debounce_9 :=TPInst9.Q;
0099  (*Button 9 pressed*)
0100  IF Numpad_Debounce_9
0101  THEN
0102  no_of_push:=no_of_push+1;
0103  Var_array[no_of_push]:='9';
0104  END_IF;
0105
0106  (*Debounce button function*)
0107  TPInst0(IN := Num_Pad_0, PT:= T#5ms);
0108  Numpad_Debounce_0 :=TPInst0.Q;
0109  (*Button 0 pressed*)
0110  IF Numpad_Debounce_0
0111  THEN
0112  no_of_push:=no_of_push+1;
0113  Var_array[no_of_push]:='0';
0114  END_IF;
0115
0116  (*Debounce button function*)
0117  TPInstDot(IN := Num_Pad_Dot, PT:= T#5ms);
0118  Numpad_Debounce_Dot :=TPInstDot.Q;
0119  (*Button . pressed*)
0120  IF Numpad_Debounce_Dot
0121  THEN
0122  no_of_push:=no_of_push+1;
0123  Var_array[no_of_push]:='.';
0124  END_IF;
0125
0126  (*Putting the two arrays together in to a STRING *)
0127  Con_12:=CONCAT(Var_array[1],Var_array[2]);
0128  Con_34:=CONCAT(Var_array[3],Var_array[4]);
0129  Con_56:=CONCAT(Var_array[5],Var_array[6]);
0130  Con_78:=CONCAT(Var_array[7],Var_array[8]);
0131
0132  Con_1234:=CONCAT(Con_12,Con_34);
0133  Con_5678:=CONCAT(Con_56,Con_78);
0134
0135  (*The merge of arrays [1]-[8] becomes one... neo*)
0136  Numpad_Display:=CONCAT(Con_1234,Con_5678);
0137
0138  (*Button OK sends the numpad value out*)
0139  IF OK
0140  THEN
0141  Numpad_out:=STRING_TO_REAL(Numpad_Display);
0142  END_IF;
0143
```

Mode_Sequence

```
0001 (*The system can be in different modes which are represented by the boxes below*)
0002
0003 (*Init is the initializing box at first startup*)
0004 (*off box is where the system is when not running anything*)
0005 (*p is the p-control box*)
0006 (*pi is the pi-control box*)
0007 (*pd is the pd-control box*)
0008 (*pid is the pid-control box*)
0009 (*i_f is the manual field current control box*)
0010 (*auto is the automnatic-control box*)
0011
0012
0013 PROGRAM Mode_Sequence
0014 VAR_INPUT
0015     Read_Button_Pressed: BOOL;
0016     Last_Button_Pressed: BOOL;
0017
0018     Read_Numpad_Value: REAL;
0019     NumP_OK_Button: BOOL;
0020
0021     Save_Button: BOOL;
0022     Clear_Button: BOOL;
0023     Close_Button: BOOL;
0024     Read_Button: BOOL;
0025     read_done: INT;
0026     Saved_light: BOOL;
0027     ReadSavedStatus_Status: INT;
0028     ReadSavedStatus_p: REAL;
0029     ReadSavedStatus_i: REAL;
0030     ReadSavedStatus_d: REAL;
0031
0032 END_VAR
0033
0034 VAR_OUTPUT
0035
0036     Read_Button_Out: BOOL:=FALSE;
0037     Mode_Save_PID: BOOL;
0038     Mode_Read_PID: BOOL;
0039
0040     Mode_Delete_Data_Segment_123: BYTE;
0041     Mode_Open_Read_close_12345: BYTE;
0042
0043     STATUS: INT;
0044     Desired_Terminal_Voltage: REAL :=156;
0045     Kp_Out: REAL;
0046     Ki_Out: REAL;
0047     Kd_Out: REAL;
0048     var_Manual_If: REAL;
0049
0050
0051
0052
0053 END_VAR
0054
0055 VAR
0056
0057     StatusText: STRING; (*Text for visualizations*)
0058     Status_Ready: INT := 0;
0059
0060
0061     Numpad_Pointer: INT; (*Can have values 1,2,3,4 or 5 which determines where
0062                          [Kp_Out, Ki_Out, Kd_Out, var_Manual_If, Desired_Terminal_Voltage]
0063                          the numpad value that is typed in should be saved*)
0064
0065     var_Manual_If_slider: INT;
0066     Back: BOOL := FALSE;
0067 END_VAR
0068
0069
0070
0071
0072
0073
0074
0075
0076
0077
0078
```


0079
0080
0081
0082
0083
0084
Init
Mode0
off
Mode1
Mode2
Mode3
Mode4
Mode5
Mode6
p
pi
pd
pid
i_f
auto
Mode7
Mode8
Mode9
Mode10
Mode11
Mode12
Init

Init

```
0001 (*Determinating if back button is used or exit button, to read or not to read, that is the question *)
0002
0003 IF Back AND NOT Saved_light
0004 THEN
0005 Status:=0;
0006 ELSE
0007 Read_Button_Out:=TRUE;
0008 Status:=0;
0009 digital_output_0:=FALSE; (*turn of static SSR's when "YES" button is used from AUTO_exit*)
0010 END_IF;
```

Off

```
0001
0002  StatusText:='Choose a Controller';
0003
0004
0005  Kp_Out:=0;
0006  Ki_Out:=0;
0007  Kd_Out:=0;
0008  Status_Ready:=0;
0009  Numpad_Pointer:=0;
0010
0011  Back:=FALSE;
0012  Mode_Read_PID:=FALSE;
0013  Read_Button_Out:=FALSE;
0014  Read_Button_Pressed:=FALSE;
0015
0016
0017  IF Read_Button
0018  THEN
0019  Mode_Open_Read_close_12345 := 2;    (*Read saved file*)
0020  Mode_Read_PID:=Read_Button;
0021  END_IF;
0022  IF Close_Button
0023  THEN
0024  Mode_Open_Read_close_12345 := 5;    (*Close saved file*)
0025  Mode_Read_PID:=Close_Button;
0026  END_IF;
0027
0028
0029
0030
0031
0032
0033
0034
0035
0036
0037
0038
0039
0040
0041
0042
0043
```

p

```
0001  Kd_Out:=0;                      (*=0 for P-Controller*)
0002  StatusText:='P Controller';
0003
0004  Mode_Save_PID:=FALSE;
0005
0006  IF Clear_Button
0007  THEN
0008  Mode_Delete_Data_Segment_123 := 1;  (*Delete old saved file*)
0009  Mode_Save_PID:=TRUE;
0010  END_IF;
0011
0012  IF Save_Button
0013  THEN
0014  Mode_Delete_Data_Segment_123 := 2;  (*Save file*)
0015  Mode_Save_PID:=TRUE;
0016  END_IF;
0017
0018  IF Last_Button_Pressed OR Read_Button_Pressed
0019  THEN
0020  Kp_Out:=ReadSavedStatus_p;
0021  Ki_Out:=ReadSavedStatus_i;
0022  Kd_Out:=ReadSavedStatus_d;
0023  END_IF;
0024
0025  IF Numpad_Pointer=1 AND NumP_OK_Button             (*Set Kp value with numpad*)
0026  THEN
0027  Kp_Out:=Read_Numpad_Value;
0028  END_IF;
0029
0030  IF Numpad_Pointer=5 AND NumP_OK_Button             (*Set desired terminal voltage value with numpad*)
0031  THEN
0032  Desired_Terminal_Voltage:=Read_Numpad_Value;
0033  END_IF;
0034
0035
0036
0037
0038
0039
0040
0041
0042
```

pi

```
0001  Kd_Out:=0;                      (*=0 for PI-Controller*)
0002  StatusText:='PI Controller';
0003
0004  Mode_Save_PID:=FALSE;
0005
0006  IF Clear_Button
0007  THEN
0008  Mode_Delete_Data_Segment_123 := 1; (*Delete old saved file*)
0009  Mode_Save_PID:=Clear_Button;
0010  END_IF;
0011
0012
0013  IF Save_Button
0014  THEN
0015  Mode_Delete_Data_Segment_123 := 2; (*Save file*)
0016  Mode_Save_PID:=TRUE;
0017  END_IF;
0018
0019
0020  IF Last_Button_Pressed OR Read_Button_Pressed
0021  THEN
0022  Kp_Out:=ReadSavedStatus_p;
0023  Ki_Out:=ReadSavedStatus_i;
0024  Kd_Out:=ReadSavedStatus_d;
0025  END_IF;
0026
0027
0028  IF Numpad_Pointer=1 AND NumP_OK_Button             (*Set Kp value with numpad*)
0029  THEN
0030  Kp_Out:=Read_Numpad_Value;
0031  END_IF;
0032
0033  IF Numpad_Pointer=2 AND NumP_OK_Button             (*Set Ki value with numpad*)
0034  THEN
0035  Ki_Out:=Read_Numpad_Value;
0036  END_IF;
0037
0038
0039  IF Numpad_Pointer=5 AND NumP_OK_Button             (*Set desired terminal voltage value with numpad*)
0040  THEN
0041  Desired_Terminal_Voltage:=Read_Numpad_Value;
0042  END_IF;
0043
```

pd

```
0001  StatusText:='PD Controller';
0002
0003  Mode_Save_PID:=FALSE;
0004
0005  IF Clear_Button
0006  THEN
0007  Mode_Delete_Data_Segment_123 := 1;      (*Clear old saved file*)
0008  Mode_Save_PID:=TRUE;
0009  END_IF;
0010
0011  IF Save_Button
0012  THEN
0013  Mode_Delete_Data_Segment_123 := 2;      (*Save file*)
0014  Mode_Save_PID:=TRUE;
0015  END_IF;
0016
0017  IF Last_Button_Pressed OR Read_Button_Pressed
0018  THEN
0019  Kp_Out:=ReadSavedStatus_p;
0020  Ki_Out:=ReadSavedStatus_i;
0021  Kd_Out:=ReadSavedStatus_d;
0022  END_IF;
0023
0024
0025  IF Numpad_Pointer=1 AND NumP_OK_Button              (*Set Kp value with numpad*)
0026  THEN
0027  Kp_Out:=Read_Numpad_Value;
0028  END_IF;
0029
0030
0031  IF Numpad_Pointer=3 AND NumP_OK_Button              (*Set Kd value with numpad*)
0032  THEN
0033  Kd_Out:=Read_Numpad_Value;
0034  END_IF;
0035
0036
0037  IF Numpad_Pointer=5 AND NumP_OK_Button              (*Set desired terminal voltage value with numpad*)
0038  THEN
0039  Desired_Terminal_Voltage:=Read_Numpad_Value;
0040  END_IF;
0041
```

pid

```
0001  StatusText:='PID Controller';
0002
0003  Mode_Save_PID:=FALSE;
0004
0005  IF Clear_Button
0006  THEN
0007  Mode_Delete_Data_Segment_123 := 1; (*Clear old saved file*)
0008  Mode_Save_PID:=TRUE;
0009  END_IF;
0010
0011  IF Save_Button
0012  THEN
0013  Mode_Delete_Data_Segment_123 := 2; (*Save file*)
0014  Mode_Save_PID:=TRUE;
0015  END_IF;
0016
0017  IF Last_Button_Pressed OR Read_Button_Pressed
0018  THEN
0019  Kp_Out:=ReadSavedStatus_p;
0020  Ki_Out:=ReadSavedStatus_i;
0021  Kd_Out:=ReadSavedStatus_d;
0022  END_IF;
0023
0024
0025  IF Numpad_Pointer=1 AND NumP_OK_Button          (*Set Kp value with numpad*)
0026  THEN
0027  Kp_Out:=Read_Numpad_Value;
0028  END_IF;
0029
0030  IF Numpad_Pointer=2 AND NumP_OK_Button          (*Set Ki value with numpad*)
0031  THEN
0032  Ki_Out:=Read_Numpad_Value;
0033  END_IF;
0034
0035  IF Numpad_Pointer=3 AND NumP_OK_Button          (*Set Kd value with numpad*)
0036  THEN
0037  Kd_Out:=Read_Numpad_Value;
0038  END_IF;
0039
0040  IF Numpad_Pointer=5 AND NumP_OK_Button          (*Set desired terminal voltage value with numpad*)
0041  THEN
0042  Desired_Terminal_Voltage:=Read_Numpad_Value;
0043  END_IF;
0044
0045
```

i_f

```
0001 (*StatusText:='Manual Field Current Controller';*)
0002
0003
0004 IF Numpad_Pointer=4 AND NumP_OK_Button          (*Set the If value with numpad*)
0005 THEN
0006 var_Manual_If:=Read_Numpad_Value;
0007 END_IF;
0008
0009 var_Manual_If:=var_Manual_If_slider*54;
```

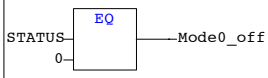
auto

```
0001
0002  Mode_Open_Read_close_12345 := 5;
0003  Mode_Read_PID:=TRUE;
0004
0005  Kp_Out:=ReadSavedStatus_p;
0006  Ki_Out:=ReadSavedStatus_i;
0007  Kd_Out:=ReadSavedStatus_d;
```

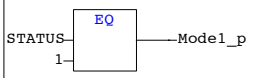
Mode

```
0001  (*Mode checks the status and determines in which mode the system is in. See Mode_Sequence*)
0002
0003  FUNCTION_BLOCK Mode
0004  VAR_INPUT
0005
0006
0007      STATUS: INT;
0008  END_VAR
0009  VAR_OUTPUT
0010      Mode1_p: BOOL;
0011      Mode3_pd: BOOL;
0012      Mode6_auto: BOOL;
0013
0014      Mode2_pi: BOOL;
0015      Mode4_pid: BOOL;
0016
0017      Mode5_IF: BOOL;
0018
0019      Mode0_off: BOOL;
0020  END_VAR
0021  VAR
0022
0023  END_VAR
```

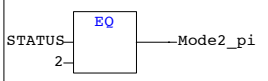
0001



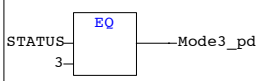
0002



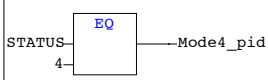
0003



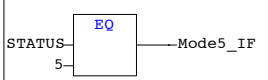
0004



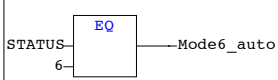
0005



0006

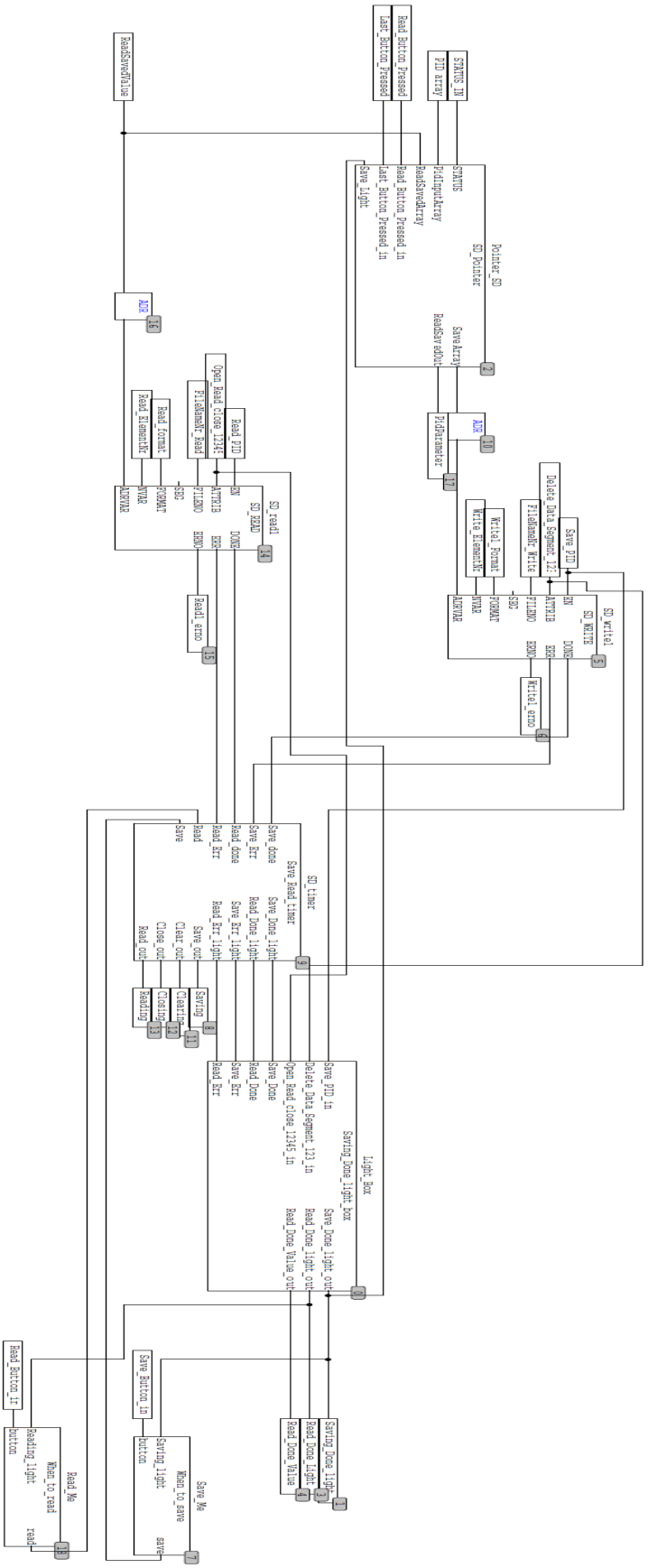


0007



SD_Save_Read

```
0001 FUNCTION_BLOCK SD_Save_Read
0002 VAR_INPUT
0003
0004     PID_array: ARRAY [1..4] OF DWORD;   (*Saving [Status, Kp, Ki, Kd] values to SD-Card*)
0005     Last_Button_Pressed: BOOL;
0006     Read_Button_Pressed: BOOL;
0007     Save_Button_in: BOOL;
0008     Read_Button_in: BOOL;
0009     Save_PID: BOOL;                     (*If true saving values*)
0010     Read_PID: BOOL;                     (*If true reading values*)
0011
0012     Delete_Data_Segment_123: BYTE;      (*1 - Delete file (Delete)
0013                                         2 - Write data set (Open(create), Write(append), Close)
0014                                         3 - Write segment label (Open(create), Write(append), Close)*)
0015
0016     Open_Read_close_12345: BYTE;    (*1 - Open file, search sector, read data set (Open, Seek, Read)
0017                                         2 - Open file, read data set (Open, Read)
0018                                         3 - Read next data set (Read)
0019                                         4 - Read data set, close file (Read, Close)
0020                                         5 - Close file (Close)*)
0021
0022     STATUS_IN: INT;
0023
0024 END_VAR
0025 VAR_OUTPUT
0026     PidParameter: ARRAY [1..4] OF DWORD;
0027     Saving: BOOL;
0028     Clearing: BOOL;
0029     Closing: BOOL;
0030     Reading: BOOL;
0031     Read_Done_Value: INT;
0032     Saving_Done_light: BOOL;
0033 END_VAR
0034 VAR
0035     Read_Done_Light: BOOL;
0036     SD_write1: SD_WRITE;
0037     SD_read1: SD_READ;
0038
0039     Write1_erno: WORD;                  (*Error number*)
0040     Read1_erno: WORD;                   (*Error number*)
0041     FileNameNr_Write: BYTE:=0;           (* Number of the storage file (XX = 0 to 99) USRDATXX.DAT
0042                                           Saving 4 different files on memory 0-3 p,pi,pd,pid*)
0043     FileNameNr_Read: BYTE:=0;            (* Number of the storage file (XX = 0 to 99) USRDATXX.DAT
0044                                         Reading 4 different files on memory 0-3 p,pi,pd,pid*)
0045     Write1_Format: BYTE := 32;          (*Data format:
0046                                         00hex - 0 - BYTE
0047                                         01 hex - 1 - CHAR
0048                                         10 hex - 16 - WORD
0049                                         11 hex - 17 - INT
0050                                         20 hex - 32 - DWORD
0051                                         21 hex - 33 - DINT*)
0052     Write_ElementNr: WORD := 13;        (*Number of elements per data set*)(*4*)
0053     Read_format: BYTE := 32;        (*Data format:
0054                                         00hex - 0 - BYTE
0055                                         01 hex - 1 - CHAR
0056                                         10 hex - 16 - WORD
0057                                         11 hex - 17 - INT
0058                                         20 hex - 32 - DWORD
0059                                         21 hex - 33 - DINT*)
0060
0061     Read_ElementNr: WORD := 13;          (*Number of elements per data set*)(*4*)
0062     ReadSavedValue: ARRAY [1..13] OF DWORD;  (*Values fron the SD-card*)(*1..4*)
0063
0064     Light_Box: Saving_Done_light_box;
0065     Pointer_SD: SD_Pointer;
0066     SD_timer: Save_Read_timer;
0067     Save_Me: When_to_save;
0068     Read_Me: When_to_read;
0069 END_VAR
```


Read_Button_Pressed
Last_Button_Pressed
STATUS_IN
PID array
ReadServoValue
Save_Light
Last_Button_Pressed_in
PidInputArray
STATUS
Read_Button_Pressed_in
Pointer_SD
SD_Pointer
Save_Array
ReadServoOut
ADDR
PidParameter
Delete_Data_Segment_123
FileNameNr_Write
Save_PID
Write_Filename
Write_Format
SD_write1
ASTRIB
SEN
FILENO
ERRNO
ERR
DONE
WNUM
FORMAT
SDC
ADDRVAR
Write_errno
Open_Read_Close_12345
FileNameNr_Read
Read_PID
Read_Filename
Read_Format
SD_read1
SD_READ
Read_errno
Save
Read
Read_Err
Save_done
Save_Err
Read_done
SD_timer
Save_Read_timer
Save_Done_light
Read_Done_light
Save_Err_light
Read_Err_light
Clear_out
Close_out
Read_out
Reading
Closing
Clearing
Saving
Save_PID_in
Delete_Data_Segment_123_in
Open_Read_close_12345_in
Light_Box
Saving_Done_light_box
Save_Done_light_out
Read_Done_light_out
Read_Done_Value_out
Read_Button_in
Reading_light
Button
When_to_read
Read_Me
Save_Button_in
Saving_light
When_to_save
Save_Me
Read_Done_Value
Read_Done_light
Saving_Done_light

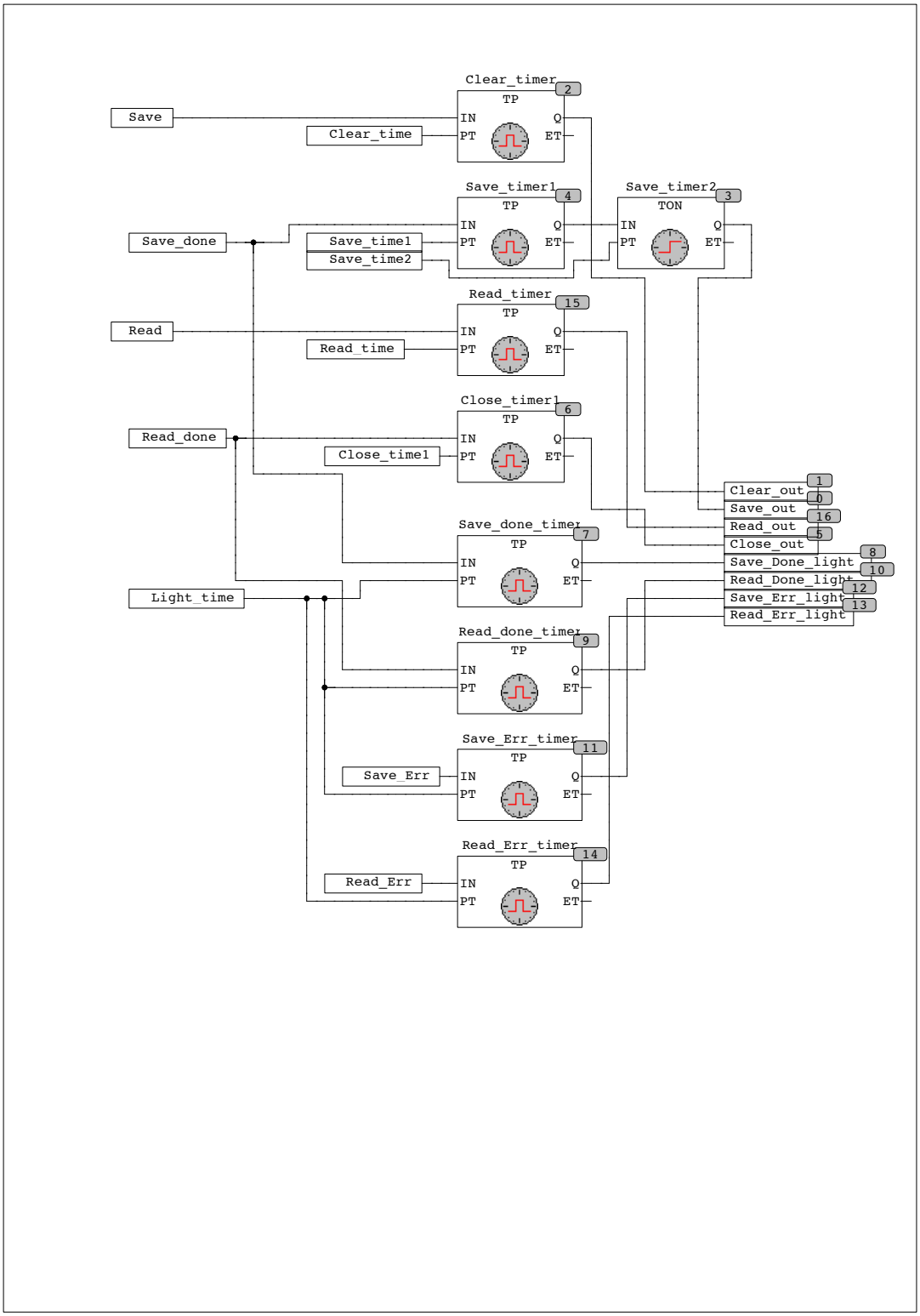
SD_Pointer

```
0001  FUNCTION_BLOCK SD_Pointer
0002  VAR_INPUT
0003      STATUS: INT;
0004      PidInputArray: ARRAY [1..4] OF DWORD;
0005      ReadSavedArray: ARRAY [1..13] OF DWORD;  (*Values from the SD-card*)
0006      Read_Button_Pressed_in: BOOL;
0007      Last_Button_Pressed_in: BOOL;
0008      Save_Light: BOOL;
0009  END_VAR
0010  VAR_OUTPUT
0011      SaveArray: ARRAY [1..13] OF DWORD;
0012      ReadSavedOut: ARRAY [1..4] OF DWORD;
0013  END_VAR
0014  VAR
0015  END_VAR
```

```
0001  ReadSavedOut[1]:=ReadSavedArray[1];
0002  SaveArray[1]:=PidInputArray[1];
0003
0004  IF STATUS=0 AND NOT Save_Light
0005  THEN
0006  SaveArray:=ReadSavedArray;
0007  END_IF;
0008
0009  IF STATUS = 1
0010  THEN
0011  IF Read_Button_Pressed_in OR Last_Button_Pressed_in
0012  THEN
0013  ReadSavedOut[2]:=ReadSavedArray[2];
0014  ReadSavedOut[3]:=ReadSavedArray[3];
0015  ReadSavedOut[4]:=ReadSavedArray[4];
0016  END_IF;
0017  SaveArray[2]:=PidInputArray[2];
0018  SaveArray[3]:=PidInputArray[3];
0019  SaveArray[4]:=PidInputArray[4];
0020  END_IF;
0021
0022  IF STATUS = 2
0023  THEN
0024  IF Read_Button_Pressed_in OR Last_Button_Pressed_in
0025  THEN
0026  ReadSavedOut[2]:=ReadSavedArray[5];
0027  ReadSavedOut[3]:=ReadSavedArray[6];
0028  ReadSavedOut[4]:=ReadSavedArray[7];
0029  END_IF;
0030  SaveArray[5]:=PidInputArray[2];
0031  SaveArray[6]:=PidInputArray[3];
0032  SaveArray[7]:=PidInputArray[4];
0033  END_IF;
0034
0035  IF STATUS = 3
0036  THEN
0037  IF Read_Button_Pressed_in OR Last_Button_Pressed_in
0038  THEN
0039  ReadSavedOut[2]:=ReadSavedArray[8];
0040  ReadSavedOut[3]:=ReadSavedArray[9];
0041  ReadSavedOut[4]:=ReadSavedArray[10];
0042  END_IF;
0043  SaveArray[8]:=PidInputArray[2];
0044  SaveArray[9]:=PidInputArray[3];
0045  SaveArray[10]:=PidInputArray[4];
0046  END_IF;
0047
0048  IF STATUS = 4
0049  THEN
0050  IF Read_Button_Pressed_in OR Last_Button_Pressed_in
0051  THEN
0052  ReadSavedOut[2]:=ReadSavedArray[11];
0053  ReadSavedOut[3]:=ReadSavedArray[12];
0054  ReadSavedOut[4]:=ReadSavedArray[13];
0055  END_IF;
0056  SaveArray[11]:=PidInputArray[2];
0057  SaveArray[12]:=PidInputArray[3];
0058  SaveArray[13]:=PidInputArray[4];
0059  END_IF;
```

Save_Read_timer

```
0001 FUNCTION_BLOCK Save_Read_timer
0002 VAR_INPUT
0003     Save_done: BOOL;
0004     Save_Err: BOOL;
0005     Read_done: BOOL;
0006     Read_Err: BOOL;
0007     Read: BOOL;
0008     Save: BOOL;
0009
0010 END_VAR
0011 VAR_OUTPUT
0012     Save_Done_light: BOOL;
0013     Read_Done_light: BOOL;
0014     Save_Err_light: BOOL;
0015     Read_Err_light: BOOL;
0016     Save_out: BOOL;
0017     Clear_out: BOOL;
0018     Close_out: BOOL;
0019     Read_out: BOOL;
0020
0021
0022 END_VAR
0023 VAR
0024
0025     (*TON The function block Timer On Delay implements a turn-on delay*)
0026     (*TP The function block is HIGH for a certain time and then returns to LOW*)
0027
0028     Close_timer1: TP;
0029
0030     Read_timer: TP;
0031     Save_timer1: TP;
0032     Save_timer2: TON;
0033     Clear_timer: TP;
0034     Save_Err_timer: TP;
0035     Read_Err_timer: TP;
0036     Save_done_timer: TP;
0037     Read_done_timer: TP;
0038
0039     Clear_time: TIME:=T#500ms;
0040     Save_time1: TIME:=T#1500ms;
0041     Save_time2: TIME:=T#500ms;
0042     Read_time: TIME:=T#50ms;
0043     Close_time1: TIME:=T#1500ms;
0044
0045     Light_time: TIME:=T#200ms;
0046
0047
0048
0049 END_VAR
0050
0051
0052
0053
0054
0055
0056
0057
0058
0059
0060
0061
0062
0063
0064
0065
0066
0067
0068
0069
```


Clear_timer
2
TP
Save
IN
Q
Clear_time
PT
ET
Save_timer1
4
TP
Save_timer2
3
TON
Save_done
IN
Q
Save_time1
PT
ET
Save_time2
Read_timer
15
TP
Read
IN
Q
Read_time
PT
ET
Close_timer1
6
TP
Read_done
IN
Q
Close_time1
PT
ET
Clear_out
1
Save_out
0
Read_out
16
Close_out
5
Save_done_timer
7
TP
Save_Done_light
8
IN
Q
Read_Done_light
10
Light_time
PT
ET
Save_Err_light
12
Read_Err_light
13
Read_done_timer
9
TP
IN
Q
PT
ET
Save_Err_timer
11
TP
Save_Err
IN
Q
PT
ET
Read_Err_timer
14
TP
Read_Err
IN
Q
PT
ET

Save_Done_light_Box

```
0001 FUNCTION_BLOCK Saving_Done_light_box
0002 VAR_INPUT
0003     Save_PID_in: BOOL;
0004     Delete_Data_Segment_123_in: BYTE;
0005     Open_Read_close_12345_in: BYTE;
0006     Save_Done: BOOL;
0007     Read_Done: BOOL;
0008     Save_Err: BOOL;
0009     Read_Err: BOOL;
0010 END_VAR
0011 VAR_OUTPUT
0012     Save_Done_light_out: BOOL;
0013     Read_Done_light_out: BOOL;
0014     Read_Done_Value_out: INT;
0015 END_VAR
0016 VAR
0017     SavingText: STRING; (*Text for visualizations*)
0018     num: INT:=0;
0019     ReadingText: STRING;    (*Text for visualizations*)
0020 END_VAR
```

```
0001 IF num=0
0002 THEN
0003     SavingText:='Save?';
0004     Save_Done_light_out:=FALSE;
0005 END_IF;
0006
0007 IF Delete_Data_Segment_123_in=1 AND Save_PID_in
0008 THEN
0009     num:=1;
0010     SavingText:='Saving..';
0011 END_IF;
0012
0013 IF Delete_Data_Segment_123_in=1 AND NOT Save_PID_in
0014 THEN
0015     SavingText:='Saving...';
0016 END_IF;
0017
0018 IF Delete_Data_Segment_123_in=2 AND num=1 AND Save_Done AND NOT Save_Err
0019 THEN
0020     Save_Done_light_out:=TRUE;
0021     SavingText:='Saved';
0022     Read_Done_light_out:=FALSE; (*We are able to read only if a new value is saved*)
0023 END_IF;
0024
0025 IF Delete_Data_Segment_123_in=2 AND num=1 AND Save_Done AND Save_Err
0026 THEN
0027     Save_Done_light_out:=FALSE;
0028     SavingText:='Error';
0029 END_IF;
0030
0031 IF Open_Read_close_12345_in=2
0032 THEN
0033     Read_Done_light_out:=FALSE;
0034     Read_Done_Value_out:=0;
0035 END_IF;
0036
0037 IF Open_Read_close_12345_in=5 AND Read_Done AND NOT Read_Err
0038 THEN
0039     Read_Done_light_out:=TRUE;
0040     Read_Done_Value_out:=1;
0041 END_IF;
0042
0043 IF Open_Read_close_12345_in=5 AND Read_Done AND Read_Err
0044 THEN
0045     Read_Done_light_out:=FALSE;
0046     Read_Done_Value_out:=0;
0047     ReadingText:='SD Error';
0048     Read_Done_Value_out:=1;
0049 END_IF;
0050
0051 (*Visualization of reading SD card*)
0052 IF Read_Done_light_out
0053 THEN
0054     ReadingText:='SD Done';
0055 ELSE
0056     ReadingText:='Reading SD';
0057 END_IF;
```

When_to_save

```
0001  FUNCTION_BLOCK When_to_save
0002  VAR_INPUT
0003      Saving_light: BOOL;
0004      button: BOOL;
0005  END_VAR
0006  VAR_OUTPUT
0007      save: BOOL;
0008  END_VAR
0009  VAR
0010  END_VAR
```

```
0001  IF  Saving_light
0002  THEN
0003  save:=FALSE;
0004  ELSE
0005  save:=button;
0006  END_IF;
```

When_to_read

```
0001 FUNCTION_BLOCK When_to_read
0002 VAR_INPUT
0003     Reading_light: BOOL;
0004     button: BOOL;
0005 END_VAR
0006 VAR_OUTPUT
0007     read: BOOL;
0008 END_VAR
0009 VAR
0010 END_VAR
```

```
0001 IF Reading_light
0002 THEN
0003 read:=FALSE;
0004 ELSE
0005 read:=button;
0006 END_IF;
0007
```

Static_or_Rotating

```
0001 FUNCTION_BLOCK Static_or_Rotating
0002 VAR_INPUT
0003     Run_in: BOOL;
0004     Static_in: BOOL;
0005     Rotating_in:BOOL;
0006     Bluetooth_in: BOOL;
0007     Arduino_Power: BOOL;
0008 END_VAR
0009 VAR_OUTPUT
0010     Run_out: BOOL;
0011 END_VAR
0012 VAR
0013     ExcitationText: STRING;
0014     ExcitationTextCollor: BOOL;
0015 END_VAR
```

```
0001 digital_output_0:=FALSE; (*ON/OFF controlling pin for the SSR's *)
0002 IF Run_in=FALSE
0003 THEN
0004 Run_out:=FALSE;
0005 END_IF;
0006
0007 (*Static Excitation*)
0008 IF Static_in
0009 THEN
0010
0011 IF Arduino_Power=FALSE (*FALSE=Arduino is ON    ,    TRUE=Arduino is OFF*)
0012 THEN
0013 ExcitationText:='Turn off Arduino, disconnect USB cable!';
0014 ExcitationTextCollor:=FALSE; (*Red*)
0015 Run_out:=FALSE;
0016 digital_output_0:=FALSE; (*ON/OFF controlling pin for the SSR's *)
0017 END_IF;
0018
0019 IF Arduino_Power=TRUE  (*True=OFF False=ON*)
0020 THEN
0021 ExcitationText:='Static Excitation ENABLED';
0022 ExcitationTextCollor:=TRUE; (*Green*)
0023 digital_output_0:=TRUE; (*ON/OFF controlling pin for the SSR's *)
0024 Run_out:=Run_in;
0025 END_IF;
0026
0027 END_IF;
0028
0029 (*Rotating Excitation*)
0030 IF Rotating_in
0031 THEN
0032 digital_output_0:=FALSE;(*ON/OFF controlling pin for the SSR's, True=ON False=OFF *)
0033
0034 IF Bluetooth_in=TRUE  (*True=OFF False=ON*)
0035 THEN
0036 ExcitationText:='Rotating Excitation, Bluetooth connection DISABLED';
0037 ExcitationTextCollor:=FALSE; (*Red*)
0038 Run_out:=Run_in; (*If bluetooth connection is lost*)
0039
0040 IF Arduino_Power  (*if power is lost*)
0041 THEN
0042 ExcitationText:='Arduino is DISCONNECTED';
0043 ExcitationTextCollor:=FALSE; (*Red*)
0044 Run_out:=FALSE;
0045 END_IF;
0046
0047 END_IF;
0048
0049 IF Bluetooth_in=FALSE  (*True=OFF False=ON*)
0050 THEN
0051 ExcitationText:='Rotating Excitation ENABLED';
0052 ExcitationTextCollor:=TRUE; (*Green*)
0053 Run_out:=Run_in;
0054 END_IF;
0055 END_IF;
0056
0057
0058
```

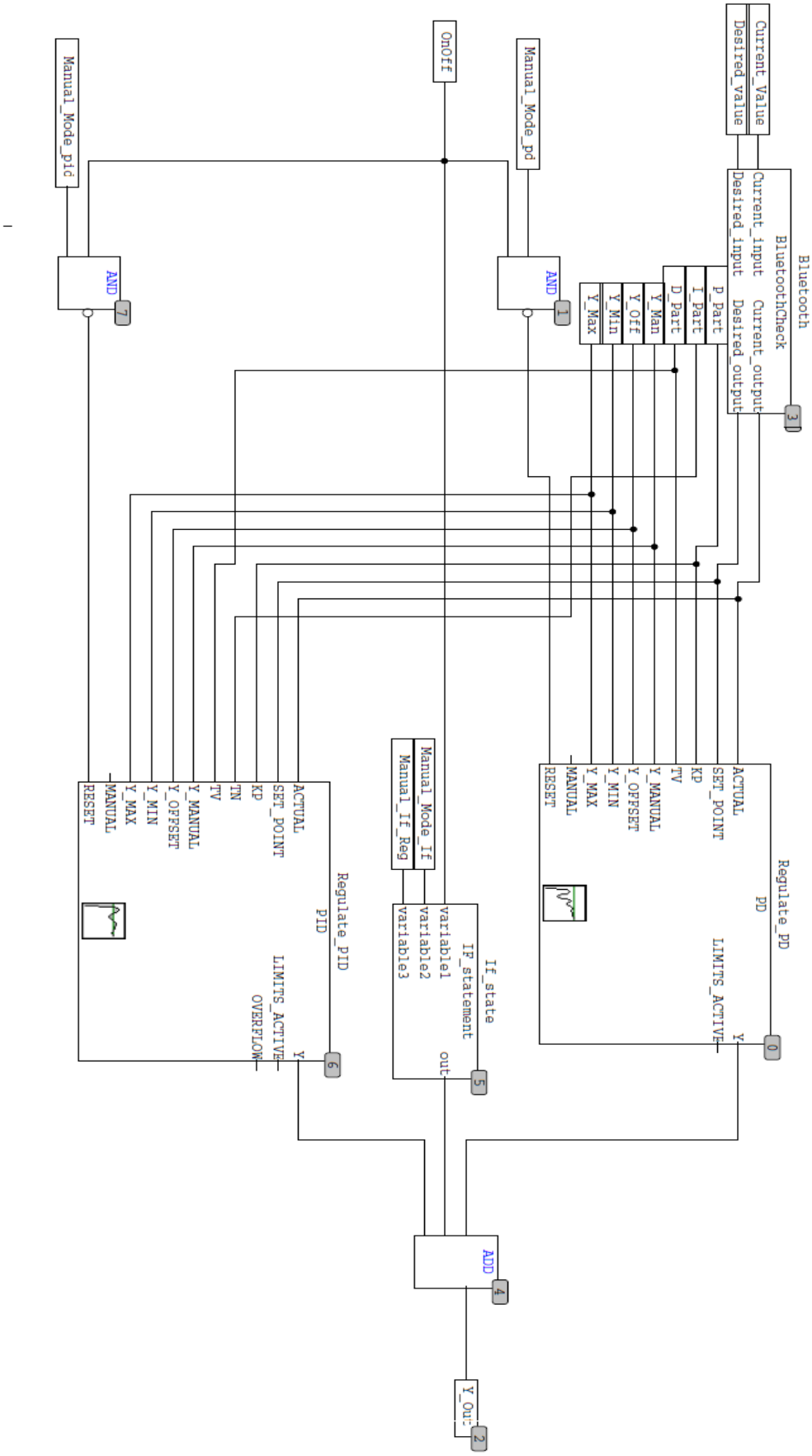
KiKd_to_TnTv

```
0001 (*Converts the Ki and Kd parameters to Tn and Tv so that they are compatible with the regulators*)
0002 FUNCTION_BLOCK KiKd_to_TnTv
0003 VAR_INPUT
0004     Kp_in: REAL;
0005     Ki: REAL;
0006     Kd: REAL;
0007 END_VAR
0008 VAR_OUTPUT
0009     Kp_out: REAL;
0010     Tn: REAL;
0011     Tv: REAL;
0012 END_VAR
0013 VAR
0014 END_VAR
```

```
0001 Kp_out:=Kp_in;
0002 Tn:=Kp_in/Ki;
0003 Tv:=Kd/Kp_in;
0004
```

Regulator

```
0001 FUNCTION_BLOCK Regulator
0002 VAR_INPUT
0003
0004     Current_Value: REAL;    (*Current value of the controlled variable*)
0005
0006
0007     P_Part: REAL;           (*Proportionality coefficient, unity gain of the P-part*)
0008     I_Part: REAL;           (*Reset time, reciprocal unity gain of the I-part;given in seconds,
0009                               e.g. "0.5" for 500 msec*)
0010     D_Part: REAL;           (*Derivative action time, unity gain of the D-part  in seconds,
0011                               e.g. "0.5" for 500 msec*)
0012
0013
0014     Desired_Value: REAL;(*Desired value, command variable*)
0015     Manual_If_Reg: REAL;(*Manual Field Current value*)
0016
0017     Manual_Mode_pd: BOOL;   (*Controls if PD-controller is used*)
0018     Manual_Mode_pid: BOOL;  (*Controls if PID-controller is used*)
0019
0020     Manual_Mode_If: BOOL;   (*Controls if manual field current is used*)
0021     Manual_Mode_off: BOOL;  (*Not in use, just visual help*)
0022
0023     OnOff: BOOL;    (*On/Off for the Controllers PD and PID*)
0024
0025
0026
0027 END_VAR
0028 VAR_OUTPUT
0029     Y_Out: REAL;        (*Manipulated value, calculated by the function block
0030                             (Y=KP×(D+1/TN[Integral]edt+TVdD/dt)+Y_OFFSET)*)
0031 END_VAR
0032 VAR
0033     Regulate_PID: PID;      (*Used for PI and PID regulation*)
0034     Regulate_PD: PD;    (*Used for P and PD regulation*)
0035     If_state: IF_statement; (*When variable1 and 2 are TRUE then output= varaible3*)
0036
0037     Y_Man: REAL;        (*Defines output value Y in case of MANUAL = TRUE*)
0038     Y_Off: REAL;        (*Offset for the manipulated variable Y*)
0039     Y_Min: REAL:= 0;    (*Minimum value for Y[out]*)
0040     Y_Max: REAL := 13824;   (* [13824=5V] [27648=10V] Lower resp (Y_Min), upper limit(Y_Max)
0041                              for the manipulated variable Y. If Y exceeds these limits,
0042                              output LIMITS_ACTIVE will be setto TRUE and Y will be kept
0043                              within the prescribed range. This control will only work if
0044                              Y_MIN<Y_MAX*)
0045
0046
0047     Bluetooth: BluetoothCheck;
0048
0049 END_VAR
0050
0051
0052
0053
0054
0055
0056
0057
0058
0059
0060
0061
0062
```


Current_Value
Desired_value
Current_input
Desired_input
BluetoothCheck
Current_output
Desired_output
Bluetooth
3
P_Part
I_Part
D_Part
Y_Man
Y_Off
Y_Min
Y_Max
OnOff
Manual_Mode_pd
Manual_Mode_pid
AND
1
AND
7
MANUAL
Y_MAX
Y_MIN
Y_OFFSET
Y_MANUAL
TV
KP
SET_POINT
ACTUAL
RESET
Regulate_PD
PD
LIMITS_ACTIVE
Y
0
Manual_Mode_If
Manual_If_Reg
variable1
variable2
variable3
IF_statement
out
If_state
5
MANUAL
Y_MAX
Y_MIN
Y_OFFSET
Y_MANUAL
TV
TN
KP
SET_POINT
ACTUAL
RESET
Regulate_PID
PID
LIMITS_ACTIVE
OVERFLOW
Y
6
ADD
4
Y_Out
2

BluetoothCheck

```
0001  FUNCTION_BLOCK BluetoothCheck
0002  VAR_INPUT
0003      Current_input: REAL;
0004      Desired_input: REAL;
0005  END_VAR
0006  VAR_OUTPUT
0007      Current_output: REAL;
0008      Desired_output: REAL;
0009  END_VAR
0010  VAR
0011      var1: REAL;
0012  END_VAR
```

```
0001  (*Bluetooth enabled*)
0002  IF digital_input_0 = FALSE
0003  THEN
0004  Current_output:=Current_input;
0005  Desired_output:=Desired_input;
0006  var1:=Desired_input;
0007  END_IF;
0008
0009  (*Bluetooth dissabled*)
0010  IF digital_input_0 = TRUE
0011  THEN
0012  Current_output:=var1;
0013  Desired_output:=var1;
0014  END_IF;
```

IF_Statement

```
0001 (*IF_statement controlls the manual control of the field current*)
0002
0003 FUNCTION_BLOCK IF_statement
0004 VAR_INPUT
0005     variable1: BOOL;    (*OnOff*)
0006     variable2: BOOL;    (*Manual_Mode_If*)
0007     variable3: REAL;    (*Manual_If_Reg*)
0008 END_VAR
0009 VAR_OUTPUT
0010     out: REAL;              (*Output value*)
0011 END_VAR
0012 VAR
0013
0014 END_VAR
```

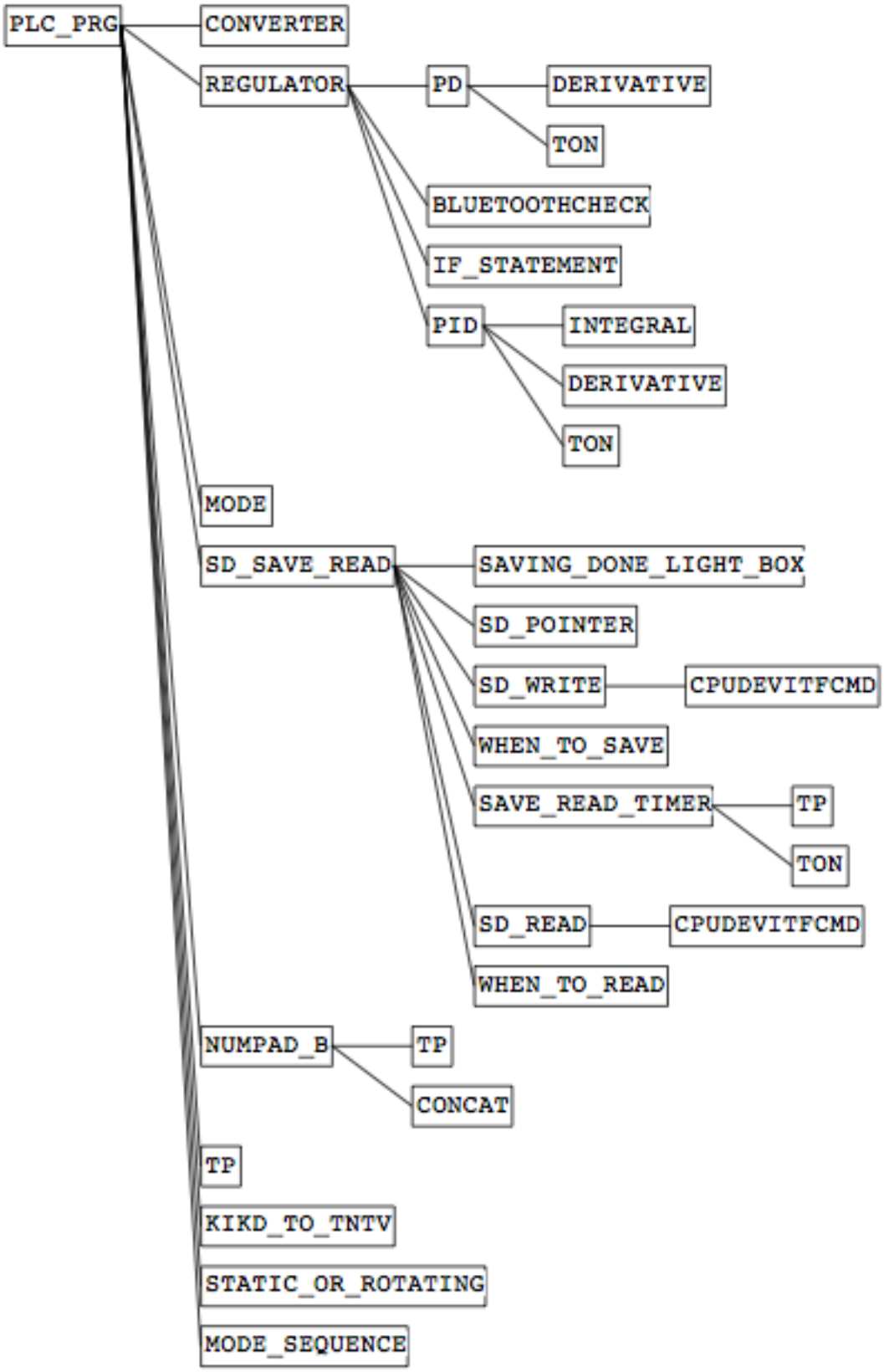
```
0001 out := 0;
0002
0003 IF variable1 AND variable2
0004 THEN
0005 out := variable3;
0006
0007 IF variable3>13824 (*13824 = 5V = Arduino maximum*)
0008 THEN
0009 out:=13824;
0010
0011 END_IF;
0012 END_IF;
0013
```

Converter

```
0001  FUNCTION_BLOCK Converter
0002  VAR_INPUT
0003
0004      in_1: ARRAY[1..4] OF DWORD;     (*ReadSavedValue*)
0005      in_2: REAL;     (*analog_input_1 Measured field current*)
0006      in_3: DWORD;                (*Status*)
0007      in_4: REAL;     (*Desired Terimnal Voltage*)
0008      in_5: REAL;     (*var_P Kp*)
0009      in_6: REAL;     (*var_I Ki*)
0010      in_7: REAL;     (*var_D Kd*)
0011      in_8: REAL;     (*analog_input_0 / Current_Value*)
0012      in_9: REAL;     (*Truncated value from PID controller*)
0013
0014  END_VAR
0015  VAR_OUTPUT
0016      out_5:ARRAY[1..4] OF DWORD;     (*PID_Array*)
0017      out_1: REAL;                (*Tillquist (VoltageTransducer) meassured Voltage*)
0018      out_2: REAL;                (*Control Voltage to SSR [Vcc]*)
0019      out_3: REAL;                (*Meassured Field Current (CurrentTransducer)*)
0020      out_4: REAL;                (*Desired  voltage*)
0021
0022  (*out_6 to 9 is the loaded status, p, i and d value*)
0023      SD_Status_out_6: INT;
0024      SD_p_out_7: REAL;
0025      SD_i_out_8: REAL;
0026      SD_d_out_9: REAL;
0027  END_VAR
0028
0029  VAR
0030      Kp: DWORD;
0031      Ki: DWORD;
0032      Kd: DWORD;
0033      DtoReal1: REAL;
0034      DtoReal2: REAL;
0035      DtoReal3: REAL;
0036      var1: REAL;
0037      var3: REAL;
0038  END_VAR
```

```
0001  (*Voltage Transducer, reading real time AC terminal voltage*)
0002  out_1 := (in_8/27648)*250;
0003
0004  (*Regulator output signal from PLC=>Arduino master=>Arduino slave=>control signal to the SSR's 0-5VDC*)
0005  out_2 := (in_9/27648)*10;
0006
0007  (*Current Transducer*)
0008  IF in_2<96      (*calibration*)
0009  THEN
0010  out_3:=0;
0011  ELSE
0012  var1 := (REAL_TO_INT(in_2/100)); (*Rounding 1.5=>2, 1.4=>1   1A=221*)
0013  var3 := (var1/2); (*Divided with 2*)
0014  out_3 := var3;
0015  END_IF;
0016
0017  (*Voltage Transducer*)
0018  out_4 := ((in_4/250)*27648);
0019
0020  (*REAL value multiplyer. Truncate to limit the amount of decimals*)
0021  Kp := TRUNC(in_5*10000000);
0022  Ki := TRUNC(in_6*10000000);
0023  Kd := TRUNC(in_7*10000000);
0024
0025  (*Different parts of the array, [Status, Kp, Ki, Kd] *)
0026  out_5[1]:=in_3;
0027  out_5[2]:=Kp;
0028  out_5[3]:=Ki;
0029  out_5[4]:=Kd;
0030
0031  (*Convert the Saved array to real constants*)
0032  DtoReal1:=in_1[2];
0033  DtoReal2:=in_1[3];
0034  DtoReal3:=in_1[4];
0035  SD_Status_out_6:=TRUNC(in_1[1]);
0036  SD_p_out_7:=(DtoReal1/10000000);
0037  SD_i_out_8:=(DtoReal2/10000000);
0038  SD_d_out_9:=(DtoReal3/10000000);
```

Call Tree of PLC_PRG



Arduino

Master

```
/*Program for Master unit*/

#include <SoftEasyTransfer.h>     //BT library
#include        <SoftwareSerial.h>//SoftwareSerial library to create a Serial connectio
#define            RxD            6//blueToothSerial receiving pin
#define            TxD            7//blueToothSerial transmitting pin
#define       PLCvoltageIn       A0//Reads the analog out from the plc on analog in A0
#define       RotorCurrent       11//The meassured current to the plc (through PWM and LI
#define     ConnectionStatus      8//Sets a digital LOW on pin8 when the bluetooth conne
#define      PackageDeliverd      9//Output pin for bt package status
#define           Power           10//Master Arduino Power
#define           Light            3//Light pin
int             PLCvoltageLocal=0;//Store the output from the PLC in PLCvoltageLocal
int                  Confirmation;//Stores confirmation voltage sent back from the slav
int           FieldCurrentLocal=0;//Store the value of FieldCurrent sent from the slav
int FieldCurrentConstrained;
float volttest;
int voltmax=255;
int lightstep=0;

SoftwareSerial    blueToothSerial(RxD,    TxD);//blueToothSerial connected to RxD and Tx
SoftEasyTransfer       ETin,            ETout;//In/out object

/*-----------------------------------------------  Structures  -----------------------

/*-----------------------------------------------  Sending  --------------------------

struct SEND_DATA_STRUCTURE{

   int                    PLCvoltage;//Outgoing Voltage
};

SEND_DATA_STRUCTURE          SendData;//Group name SendData

/*-----------------------------------------------  Receiving  ------------------------

struct RECEIVE_DATA_STRUCTURE{

  int          PLCvoltageconfirmation;//Incomming confirmation
  int                    FieldCurrent;//Incomming Field current data
};


RECEIVE_DATA_STRUCTURE    ReceiveData;//Group name ReceiveData

/*-------------------------------------------- End of structures  --------------------



void setup()
{

  Serial.begin(9600);                                                  // Open serial
  blueToothSerial.begin(9600);
```

```
  ETin.begin(details(ReceiveData),                    &blueToothSerial);//Start the b
  ETout.begin(details(SendData), &blueToothSerial);

  // Input and outputs
  pinMode(RxD, INPUT);
  pinMode(TxD, OUTPUT);
  pinMode(PLCvoltageIn, INPUT);
  pinMode(ConnectionStatus, OUTPUT);
  pinMode(RotorCurrent, OUTPUT);
  pinMode(PackageDeliverd, OUTPUT);
  pinMode(Light,OUTPUT);
  pinMode(Power,OUTPUT);
//  pinMode(statusPin, INPUT);
}


void Status(void)
{

while (!blueToothSerial.available())                                  // Stay inside

    {
        digitalWrite(ConnectionStatus, LOW);
        Serial.println("NO BT");
        Breathing();
   }
analogWrite(Light,255);

   //volttest = ((5*ReceiveData.FieldCurrent)/1024.0000);//-3.91)*52;
    // Serial.print(ReceiveData.FieldCurrent); Serial.print("       ");     Serial.print
    //Serial.println(analogRead(PLCvoltageIn));
     digitalWrite(ConnectionStatus, HIGH);

      PLCvoltageLocal =map(analogRead(PLCvoltageIn),  0,  1023,  0,  255);//Read the ou
                                                          BTtransfer();//Start BT t
    analogWrite(RotorCurrent,FieldCurrentLocal);                        //Send the fi
    //    analogWrite(RotorCurrent,voltmax);                          //Send the field
     if(Confirmation  ==  PLCvoltageLocal){
       digitalWrite(PackageDeliverd, LOW);
     }
       else {
       digitalWrite(PackageDeliverd, HIGH);

     }

}

void BTtransfer (void)
{
 /*-----------------------------------------------  BT-transfer out  -----------------

   ETout.sendData();         //Send the data out
   SendData.PLCvoltage = PLCvoltageLocal;
```

```
 /*---------------------------------------------  BT-transfer in  -------------------

   for(int i=0; i<5; i++){
   ETin.receiveData();      //Read the incomming data
   delay(10);}

   Confirmation = ReceiveData.PLCvoltageconfirmation;
    FieldCurrentConstrained =constrain(ReceiveData.FieldCurrent,0,1023);
    FieldCurrentLocal =map(FieldCurrentConstrained, 0, 1023, 0, 255);
    }



/* ------------------------------------------------Debbuging program-------------------


void Breathing (void)
{
    for(lightstep = 0 ; lightstep <= 255; lightstep+=1)
  {

    analogWrite(Light, lightstep);
    if (lightstep > 150) {
      delay(4);
    }
    if ((lightstep > 125) && (lightstep < 151)) {
      delay(5);
    }
    if (( lightstep > 100) && (lightstep < 126)) {
      delay(7);
    }
    if (( lightstep > 75) && (lightstep < 101)) {
      delay(10);
    }
    if (( lightstep > 50) && (lightstep < 76)) {
      delay(14);
    }
    if (( lightstep > 25) && (lightstep < 51)) {
      delay(18);
    }
    if (( lightstep > 1) && (lightstep < 26)) {
      delay(19);
    }
  }

  for(lightstep = 255; lightstep >=0; lightstep-=1)
  {
    analogWrite(Light, lightstep);
    if (lightstep > 150) {
      delay(4);
    }
    if ((lightstep > 125) && (lightstep < 151)) {
      delay(5);
    }
    if (( lightstep > 100) && (lightstep < 126)) {
```

```
      delay(7);
    }
    if (( lightstep > 75) && (lightstep < 101)) {
      delay(10);
    }
    if (( lightstep > 50) && (lightstep < 76)) {
      delay(14);
    }
    if (( lightstep > 25) && (lightstep < 51)) {
      delay(18);
    }
    if (( lightstep > 1) && (lightstep < 26)) {
      delay(19);
       }
}
  }



void Debugging (void)
{

int DebuggPLCvoltagecheck = 1;
float Voltage = 0;
int PLCvoltagecheck = 0;


//if (ConnectionStatus == HIGH){

  //if (PLCvoltagecheck > SendData.PLCvoltage + DebuggPLCvoltagecheck || PLCvoltagechecl
  Voltage = (SendData.PLCvoltage*5.0)/255.0;
  float ConfirmationVoltage = (Confirmation*5.0)/255.0;
  Serial.println("Master");
  Serial.print("PLCvoltage data:  "); Serial.println(PLCvoltageLocal);
  Serial.print("Sending PLCvoltage:  "); Serial.print(Voltage); Serial.println(" V");
  Serial.print("Voltage confirmation:  "); Serial.print(ConfirmationVoltage); Serial.pr
 //Serial.print(analogRead(PLCvoltageIn));
  Serial.print("Field current data:  "); Serial.println(FieldCurrentLocal);
   Serial.println(" ");
  PLCvoltagecheck = SendData.PLCvoltage;


// else {
 //  Serial.print("Connection not established! Searching for Slave");
   //return;
//}
//}
     //  Serial.println(val);
}



void loop()
{
```

```
 digitalWrite(Power, HIGH); // Send to PLC that Arduino is on

  Status();
//  Debugging();
delay(110);//100

 }
```

Slave

```
/*Program for Slave unit*/

#include <SoftEasyTransfer.h>     //BT library
#include        <SoftwareSerial.h>//SoftwareSerial library to create a Serial connectio
#define            RxD           6//blueToothSerial receiving pin
#define            TxD           7//blueToothSerial transmitting pin
#define      SSrVoltageOut      11//Output pin connected to LP-filter due to PWM
#define       RotorCurrent      A0//Input pin that reads the rotor current (in voltage)
int                  SSrVoltage=0;//Store the regulated value in SSrVoltage
int           FieldCurrentLocal=0;//Store the value from input pin RotorCurrent
float volttest;

SoftwareSerial  blueToothSerial(RxD,  TxD);//blueToothSerial connected to RxD and TxD
SoftEasyTransfer      ETin,         ETout;//In/out object









/*-----------------------------------------------  Structures  ------------------------

/*-----------------------------------------------  Sending  --------------------------
struct SEND_DATA_STRUCTURE{

   int          PLCvoltageconfirmation;//Outgoing confirmation
   int                    FieldCurrent;//Outgoing Field current data
};

SEND_DATA_STRUCTURE           SendData;//Group name SendData

/*-----------------------------------------------  Receiving  ------------------------

struct RECEIVE_DATA_STRUCTURE{

    int                     PLCvoltage;//Incomming voltage
  };


RECEIVE_DATA_STRUCTURE     ReceiveData;//Group name ReceiveData

/*-----------------------------------------------  End of structures  ----------------


void setup()
{

  Serial.begin(9600);                                  //Open serial communications an
  blueToothSerial.begin(9600);

  ETin.begin(details(ReceiveData),     &blueToothSerial);//Start the bt library, data d
```

```
  ETout.begin(details(SendData), &blueToothSerial);

  //Input and outputs
  pinMode(RxD, INPUT);
  pinMode(TxD, OUTPUT);
  pinMode(SSrVoltageOut, OUTPUT);
  pinMode(RotorCurrent, INPUT);
  //pinMode(10, OUTPUT);

  }


void Status(void)
{
/*
while (!blueToothSerial.available())

      {
               analogWrite(SSrVoltageOut,SSrVoltage);

 }*/

      // FieldCurrentLocal = map(analogRead(RotorCurrent), 0, 1023, 0, 255);   //Store
       BTtransfer();
       analogWrite(SSrVoltageOut,SSrVoltage);        //Regualated output (PWM)

       //Serial.println(ReceiveData.PLCvoltage);
      FieldCurrentLocal=analogRead(RotorCurrent);
     //  Serial.println(FieldCurrentLocal);


}

void BTtransfer (void)
{

 /*-----------------------------------------------  BT-transfer out  -----------------

  ETout.sendData();                       //Send the data out
  SendData.FieldCurrent = FieldCurrentLocal;
  SendData.PLCvoltageconfirmation = ReceiveData.PLCvoltage;

 /*-----------------------------------------------  BT-transfer in  ------------------
   for(int i=0; i<5; i++){
   ETin.receiveData();                    //Read the incomming data
   delay(10);}

     SSrVoltage  =  ReceiveData.PLCvoltage;//Store the incomming PLCvoltage in the varial
}



/* ------------------------------------------------Debbuging program-------------------
```

```
void Debugging (void)
{
int PLCvoltagecheck = 0;
int DebuggPLCvoltagecheck = 1;
  float Volt = (ReceiveData.PLCvoltage*5.0)/255.0;
  float ConfirmationVoltage = (SendData.PLCvoltageconfirmation*5.0)/255.0;
  Serial.println("Slave");
  Serial.print("Receiving PLCvoltage data:  "); Serial.println(SSrVoltage);
  Serial.print("Receiving PLCvoltage:  "); Serial.print(Volt); Serial.println(" V");
  Serial.print("Voltage confirmation:  "); Serial.print(ConfirmationVoltage); Serial.pr
  Serial.print("Field current data:  "); Serial.println(FieldCurrentLocal);
  Serial.println(" ");
  //Serial.flush();
    PLCvoltagecheck = ReceiveData.PLCvoltage;
 // }
}
/* ------------------------------------------------End of Debbuging --------------------

void loop()
{

  Status();
//Debugging();

 delay(50);
}
```